MASTER EN SCIENCES INFORMATIQUES

Les mesures de la qualité et de la productivité des logiciels

Collin, Laurent

Award date:
1995

Awarding institution:
Universite de Namur

FACULTES
UNIVERSITAIRES
N.D. DE LA PAIX

NAMUR

INSTITUT D'INFORMATIQUE

LES MESURES DE LA QUALITE ET DE LA PRODUCTIVITE DES LOGICIELS.

Promoteur
Naji HABRA

Mémoire présenté par
Laurent Collin
en vue de l'obtention
du titre de
licencié et maître en informatique.

Année académique 1994-1995.

Résumé

La qualité d'un logiciel est une notion difficile à approcher. C'est pourquoi actuellement il y a peu de mesures fiables et précises de qualité.
Dans le cadre de ce mémoire, des métriques relatives aux attributs suivants d'un logiciel seront présentées: la taille du code, la structure logique, la structure de données, la modularité, la "reliability", la maintenabilité et l'utilité.
Les modèles d'effort et de productivité qui seront proposés sont d'une grande utilité pour estimer différents paramètres comme le coût, la date de livraison, etc.
Datrix, un outil générant des métriques, sera présenté.
Notons enfin que des outils, des techniques et un département de SQA peuvent permettre, comme nous le verrons, d'améliorer la qualité d'un logiciel.

Abstract

Software quality is a difficult notion to define. So, actually very few precise and reliable measurements of quality are available.
In this thesis, metrics about the following attributes will be presented: size of code, logical structure, data structure, modularity, reliability, maintainability and usability.
The effort and productivity models, which will be proposed, are very useful to estimate different parameters as cost, delivery date, etc.
Datrix, a tool generating metrics, will be presented.
Tools, techniques and a department of SQA can ameliorate, as we will see, the software quality.

Remerciements.

Qu'il me soit permis d'exprimer toute ma gratitude à Monsieur N. HABRA pour avoir accepté d'être le promoteur de ce mémoire. Son accueil bienveillant, ses judicieux conseils furent d'un précieux recours dans la rédaction de cet ouvrage.

Je remercie Monsieur P. Robillard, professeur à l'École Polytechnique de Montréal, pour son accueil et son dévouement lors de mon stage.

Je tiens également à remercier mes parents pour leur soutien.

TABLE DES MATIÈRE.

INTRODUCTION

Le "software engineering" a pour but d'augmenter la qualité et la productivité des logiciels. En effet, trop souvent des logiciels sont rejetés ou délivrés en retard, dépassant les budgets et de mauvaise qualité.

Même si un logiciel est reçu à temps, respectant le budget et exécutant correctement les fonctions demandées, il se peut qu'il ne soit pas satisfaisant par manque de qualité:

- Le logiciel peut être difficile à comprendre et à modifier, ce qui entraînera des coûts énormes de maintenance (la phase de maintenance représente environ 60% du budget de développement).

- Le logiciel peut être difficile à utiliser et alors inciter aux erreurs. Là encore, des coûts énormes peuvent être générés.

Une attention à la qualité d'un logiciel peut permettre d'économiser beaucoup. C'est d'autant plus important que le domaine du logiciel est en crise: les coûts du hardware baissent très vite alors que les coûts de logiciel ne cessent d'augmenter et deviennent exorbitants. Les chiffres ne pourraient être plus explicites: en 1955, le ratio des coûts du logiciel par rapport au hardware était de 15/85. En 1985, ce ratio était de 85/15.

La notion de qualité n'a pas toujours suscité autant d'intérêt dans l'industrie du logiciel (il en est de même dans l'industrie en général). Dans le passé, il était courant de penser dans les laboratoires de développement que coût et qualité étaient en conflit. On favorisait alors les coûts au dépend de la qualité. Fort heureusement, des changements sont intervenus:

- Le secteur du développement et de la maintenance de logiciel est devenu plus compétitif: il faut désormais produire de la qualité à faible coût.

- Les applications logicielles sont de plus en plus complexes et de plus en plus présentes dans la vie de tous les jours. Les fautes des logiciels peuvent avoir des effets dévastateurs: elles peuvent menacer la vie ou encore entraîner des dommages financiers. C'est le cas par exemple des applications médicales ou de gestion du trafic aérien.

- Les clients sont devenus plus exigeants envers la qualité.

Étant convaincu de la nécessité d'avoir des logiciels de qualité, encore faut-il se mettre d'accord sur ce que signifie « qualité ». Cette notion est plutôt générale et intuitive, ce qui pose problème surtout dans le domaine scientifique. Actuellement, il n'y a pas de définition standard de la qualité.

Au chapitre 1, nous essayerons d'approcher cette notion de qualité en présentant des attributs ou caractéristiques de qualité définis par Boehm, Brown et Lipow en 1976. Les relations existantes entre ces attribut seront mises en évidence.

Pour voir si un logiciel aura la qualité souhaitée, il faut mesurer tout au long du développement les attributs jugés importants. Au chapitre 2, les métriques[1] les plus courantes seront présentées. La plupart concernent les attributs internes, c'est à dire ceux qui ne dépendent pas de l'environnement. La taille du code est un exemple d'attribut interne. Les chercheurs ont commencé à s'intéresser aux métriques dans les années 1970, quand les coûts du logiciel par rapport au hardware ont commencé à augmenter. Ils ont alors essayé d'améliorer les métriques définies auparavant dans les années 1960 (comme les mesures de la taille du code). De nouvelles métriques ont été définies (comme les mesures de la complexité) mais il reste à montrer leur consistance.
Les métriques les plus utilisées sont des mesures de la taille du code, de la structure logique, de la structure de donnée mais aussi des métriques relatives à l'effort et au coût.

Les attributs externes dépendent de l'environnement. Comme ils sont trop généraux pour être mesurés directement, ils sont quantifiés à l'aide des attributs internes. Au chapitre 3 seront présentées des métriques pour les attributs externes suivants: la maintenabilité, l'utilité et la "reliability".

Le "software engineering" cherche à évaluer la productivité, en particulier pour estimer les coûts de développement. Au chapitre 4, 2 types de modèles seront abordés: les modèles de productivité et de coûts.

Pour produire rapidement des métriques, il est nécessaire de disposer d'outils comme Datrix, présenté au chapitre 5. Datrix est un outil destiné à mesurer la qualité d'un logiciel. Il a été développé à l'école Polytechnique de Montréal et produit un ensemble de métriques à partir d'un graphe de contrôle et du code source.

Pouvoir mesurer la qualité d'un logiciel au cours de son développement n'est pas suffisant pour obtenir un logiciel de qualité. Des pratiques de programmation et de design, des techniques et des outils peuvent permettre d'améliorer le processus de développement de logiciel et ainsi obtenir une meilleure qualité. C'est ce qui nous intéressera au chapitre 6.

---

[1] une métrique est une mesure d'un attribut.

Au chapitre 7, l'activité de Software Quality Assurance (SQA) sera présentée. Son but est d'améliorer la qualité et la productivité des logiciels.

Une vue générale du problème de la mesure et de l'amélioration de la qualité aura ainsi été présentée. Nulle doute que des évolutions et améliorations auront lieu dans ce domaine qui a récemment suscité un intérêt particulier.

CHAPITRE 1:

LES CARACTÉRISTIQUES DE LA QUALITÉ.

L'objectif de ce chapitre est de définir des caractéristiques (ou attributs) pour évaluer la qualité d'un logiciel. Boehm, Brown et Lipow ont définit 11 caractéristiques globales[2] et des métriques en parallèle. Une métrique est une mesure quantitative du degré avec lequel un programme à une certaine caractéristique. Le rôle de ces métriques est de vérifier comment le logiciel possède la caractéristique associée. Nous reviendrons au chapitre 2 sur les métriques.

Donnons d'abord une définition de la qualité d'un logiciel, formulée par Boehm dans « Quantitative evaluation of software quality »:
« La qualité est la composition de tous les attributs (ou caractéristiques) qui décrivent le degré d'excellence d'un logiciel. Elle ne dépend pas de ce que l'on désire mais plutôt de la nature du logiciel. La qualité inclut tous les aspects d'un logiciel: parfois la documentation est aussi importante que le programme et les données ».

En 1.1, les caractéristiques générales définies par Boehm seront présentées. Des relations existent entre ces caractéristiques générales (voir 1.2) qui sont décomposées en attributs de plus bas niveau appelés caractéristiques secondaires ou primitives (1.3).
Pour finir, en 1.4, un commentaire sera fait sur l'arbre obtenu à partir de toutes les caractéristiques (générales et secondaires) définies.

1.1 Définition des Caractéristiques Générales (ou facteurs).

Les caractéristiques présentées ici ne se recoupent pas et sont complètes.
Voici ces 11 caractéristiques:

- la Complétude (completeness): un code est complet si toutes ses parties sont présentes et si chaque partie est développée entièrement.

[2] aussi appelées par d'autres auteurs attributs ou facteurs.

- la Compréhensibilité (understandability): un code possède cette caractéristique s'il est clair pour la personne qui l'inspecte. Par exemple, les noms de variable utilisés sont évocateurs.

- la Concision (conciseness): un code est concis s'il n'y a pas d'information excessive dans le code. Ceci implique que les programmes ne soient pas excessivement fragmentés en modules et en fonctions. Il ne faut pas non plus avoir de code répété en plusieurs endroits, alors qu'une fonction aurait pu être définie.

- la Consistance (consistency): elle se décompose en 2 sous caractéristiques:
  - la consistance interne: un logiciel est consistant de manière interne si les notations, la terminologie utilisées sont uniformes. Par exemple, il n'y aura pas de commentaires trop verbeux à un endroit alors que certaines parties manque de commentaires.
  - la consistance externe: un logiciel est consistent de manière externe s'il y a un lien entre le programme et les besoins. Par exemple, les noms des variables et leurs définitions seront consistants avec un glossaire.

- l'Efficacité (efficiency): un code est efficace si on n'utilise pas de ressources inutiles dans le code. Cela implique que le code est construit de manière à produire le moins d'objets (variables, fichiers, etc) possibles.

- la Maintenabilité (maintainability): un code est maintenable s'il est facile à mettre à jour pour satisfaire de nouveaux besoins ou pour corriger des erreurs. Le code doit être compréhensible, testable, modifiable. Par exemple des commentaires seront mis pour localiser les appels de fonctions. Ou encore il y a des formats d'indentation pour retrouver facilement les instructions de branchement (if-then-else).

- la Portabilité (portability): un système est portable si le code peut s'exécuter facilement et correctement sur d'autres configurations que celle qui est courante. Pour avoir un code portable, il ne faut alors pas utiliser des caractéristiques spéciales des langages qui ne sont pas disponibles dans d'autres configurations. Des fonctions de librairies qui ne sont pas applicables dans d'autres configurations ne sont pas appelées.

- la Reliability (traduit en français par qualité, robustesse, solidité, fiabilité): le code va exécuter les fonctions attendues de manière satisfaisante. Le programme devra compiler, charger, s'exécuter et produire des réponses de la précision requise. Le code sera complet, consistant, etc. De manière plus concise, un code possède la "reliability" s'il ne contient pas d'erreur.
  Ce terme reliability a été beaucoup repris dans la littérature et on fait souvent la confusion entre "software quality" et "software reliability".

- la Structuration (structuredness): un logiciel est structuré s'il y a une certaine organisation des parties interdépendantes du programme. Ceci implique que le design a été fait de manière ordonnée.

- la Testabilité (testability): un code est testable s'il permet d'établir facilement des critères de vérification et s'il permet l'évaluation de ses performances. La découpe en modules aura alors été faite de façon à vérifier facilement que les besoins sont satisfaits.

- l'Utilité (usability): on cherche ici à décrire comment le code est utile. Une code possède cette caractéristique s'il est efficace, human engineering (cette caractéristique secondaire est définit plus loin) et s'il possède la « reliability ». On a alors des fonctions utiles, un code robuste aux erreurs humaines et qui ne consomment pas trop de mémoire, etc.

Remarquons que d'autres auteurs ont défini aussi des caractéristiques. Par exemple, Mac Call définit 3 pôles de caractéristiques:

1. Révision: maintenabilité, modifiabilité, testabilité.
2. Opération: correction, reliability, efficacité, intégrité, utilité.
3. Transition: portabilité, réutilisation, interopérabilité.

1.2 Relations entres les Caractéristiques de Qualité.

1.2.1 Conflit entre caractéristiques.

Un des problèmes majeurs est le conflit entre des caractéristiques individuelles. Par exemple, l'efficacité est souvent au prix de la portabilité, de la compréhension, de la maintenance (en effet, pour avoir un programme efficace, on doit choisir des algorithmes qui prennent le moins de temps possible, c'est à dire qu'ils sont bien souvent compliqués à cause de l'optimisation. La maintenance sera alors dégradée). La concision est en conflit avec la lisibilité. On ne peut alors avoir toutes ces caractéristiques à la fois dans un logiciel et les utilisateurs peuvent alors avoir du mal à quantifier leurs préférences.

1.2.2 Les dépendances entre les caractéristiques.

Comme nous pouvons le constater, ces caractéristiques sont liées entre elles. Par exemple, pour qu'un programme soit maintenable, il doit être compréhensible. Boehm a alors pu construire un « arbre » de dépendance entre les caractéristiques (voir *figure 1.1*). Quand 2 caractéristiques sont reliées, celle de droite est une condition nécessaire de celle de gauche.

De plus, en analysant le concept de compréhensibilité, Boehm a constaté qu'un programme compréhensible devait être structuré, consistent, concis mais aussi lisible et auto-descriptif. Il a alors définit des caractéristiques secondaires ou primitives qui étaient des conditions nécessaires pour les caractéristiques générales.

1.3 Définition des Caractéristiques Secondaires.

Boehm en définit 12:

- l'Accessibilité (accessibility): le code est accessible s'il est construit de telle façon que l'on peut facilement sélectionner ses parties.

- l'Augmentabilité (augmentability): cette caractéristique signifie qu'il est facile d'étendre le code (d'ajouter de nouvelles fonction par exemple).

- l'Auto-description (self-descriptiveness): le code contient assez d'informations pour qu'un lecteur détermine ou vérifie ses objectifs, hypothèses, contraintes ou input/output. On a recours aux commentaires.

- la « Communicativeness » (on peut le traduire par pouvoir de communiquer): les entrées/sorties ont une forme et un contenu facile à assimiler et à utiliser.

- la Facilité à modifier (modifiability): un code est facile à modifier si est facile d'y incorporer des changements.

- « Human-engineering » (on pourrait traduire ce terme par convivialité): les utilisateurs ne perdent pas leur temps et leur énergie en utilisant le logiciel. Leur moral n'est pas dégradé non plus.

- l'Indépendance vis à vis des périphériques (device-indépendance): le code peut être exécuté sur des configurations hardwares autre que la courante. C'est évidemment une condition nécessaire pour la portabilité.

- la Lisibilité (legibility): un code est lisible si en le lisant, il est facile de discerner ce qu'il fait. Pour cela, des expressions complexes auront des noms de variables mnémoniques et des parenthèses même si ce n'est pas nécessaire. Une attention particulière est portée à la présentation.

- la Mesurabilité (accountability): cette caractéristique signifie qu'il est facile d'effectuer des mesures sur le code, comme par exemple calculer le temps d'exécution d'un segment critique.

- la Précision (accuracy): un code est précis si les outputs qu'il génère sont suffisamment précis.

- la Robustesse (robustness): un code est robuste s'il peut continuer à s'exécuter même s'il y a des violations. Par exemple, si on introduit des inputs dans un format ou un type différent de celui défini, le programme ne s'interrompra pas et s'exécutera sans dégrader les performances des fonctions qui ne dépendent pas de ces inputs non standards.

- la « Self-containedness » (que l'on peut traduire par auto suffisant): le code exécute toutes ses fonctions implicites (initialisation, vérifications des inputs, etc) et explicites lui-même.

Ces caractéristiques primitives permettent de définir des métriques utilisées pour évaluer les caractéristiques de plus haut niveau, qui dépendent des primitives.

1.4 L'Arbre des Caractéristiques de Qualité.

Au plus haut niveau de la structure de l'arbre de la *figure 1.1*[3] se trouvent les caractéristiques importantes lors d'une première évaluation d'un package.
En effet, lorsqu'on acquiert un package, on se demande:

- Comment vais-je l'utiliser? (facilement, efficacement, sans avoir d'erreur)
- Comment vais-je le maintenir facilement? (pour le maintenir, il fait le comprendre et pouvoir le modifier et le tester)
- Puis-je toujours l'utiliser si je change mon environnement?

Pour évaluer l'utilité générale, on doit connaître l'utilité, la maintenabilité, la portabilité au moins (ce sont des conditions nécessaires mais pas suffisantes: pour certains programmes qui demande un sécurité informatique par exemple, d'autres caractéristiques seront nécessaires).

L'arbre montre qu'un programme est portable s'il est indépendant vis-à-vis des périphériques et s'il possède la « self-containedness ». Le code n'a pas besoin d'être précis.

Les caractéristiques de plus bas niveau (à droite dans l'arbre) peuvent être mesurées directement et sont des attributs internes (ils ne dépendent que du produit, pas de l'environnement).

Les caractéristiques de plus haut niveau (à gauche dans l'arbre) sont des attributs externes (en plus du produit, ils dépendent de l'environnement externe). Ces caractéristiques sont sémantiquement plus complexes et sont mesurées indirectement, à partir des métriques associées aux caractéristiques de plus bas niveau.

Cette hiérarchie entre les caractéristiques générales et secondaires constitue un modèle de la qualité. Ce n'est pas le seul modèle qui existe mais c'est un des premiers et un des plus complets.

[3] C'est à dire à gauche de l'arbre.

Figure 1.1: L'arbre des dépendances des caractéristiques de qualité.

CHAPITRE 2:

LES METRIQUES.

Les attributs internes d'un logiciel sont ceux qui ne dépendent pas de l'environnement. Ils ont l'avantage qu'on peut les mesurer tôt dans le cycle de développement de logiciel. Ils sont importants pour mesurer la qualité d'un logiciel car ils permettent, comme on le verra, de mesurer les attributs externes (ceux qui nous intéressent le plus) comme la reliability. Si on s'aperçoit qu'un logiciel a de mauvaises mesures d'attributs externes, il faudra alors caractériser les attributs internes défaillants et les améliorer.

Le but de ce chapitre est de faire une synthèse des métriques les plus utilisées dans l'industrie. Les métriques qui seront présentées sont les plus employées et sont considérées comme les plus utiles par les chercheurs. La plupart concernent les attributs internes (excepté en 2.5 où on verra les métriques relatives à l'effort et au coût et en 2.6 qui abordera des métriques de reliability).
On verra entre autre des métriques de taille (2.1), de structure logique (2.2), de structure de données (2.3) et des métriques de modularité (2.4).

2.1 Les Métriques concernant la Taille.

Il est intéressant d'évaluer la taille d'un programme car c'est le facteur le plus important de beaucoup de modèles de développement de logiciel (comme les modèles d'effort et de productivité). En plus, remarquons qu'elle est facile à calculer (mais pas à estimer!).
On ne doit pas décrire la taille par un attribut unique comme la longueur (tout comme on ne peut décrire la taille d'un homme seulement à partir de son poids. On doit aussi mesurer sa hauteur). Pour bien caractériser la taille d'un produit (comme le code ou un document de design), on doit mesurer au moins la longueur, la fonctionnalité et la complexité du problème à résoudre.

Pour mesurer la taille, des métriques comme le nombre de lignes de code (2.1.1), le nombre de jetons (2.1.2) peuvent être utilisées. Si le programme contient du code réutilisé, une mesure de taille équivalente (2.1.3) fait correspondre une taille en nouvelles lignes de code au programme. Les métriques proposées en 2.1.4 permettent de prédire la longueur du code. Afin de mieux caractériser la taille, il est intéressant de connaître la fonctionnalité du système (2.1.5).

2.1.1 Les lignes de Code (Loc).

Le nombre de lignes de code est la mesure la plus familière pour mesurer la longueur. On la représente par le symbole Ss. Son unité est Loc (Line Of Code) ou KLoc.
Il n'y a pas d'accord général sur ce qu'est une ligne de code. Avec certains langages, on peut mettre 2 instructions sur la même ligne. On peut alors considérer qu'il faut compter 2 lignes même s'il n'y en a qu'une. De même, certains comptent les lignes de commentaires et les lignes blanches.

Malgré tout, la définition prédominante d'une ligne de code est la suivante:
" Une ligne de code est une ligne de texte d'un programme qui n'est pas un commentaire ou une ligne blanche. On ne tient pas compte du nombre d'instructions par ligne" .

Les lignes de code ne permettent que de mesurer la longueur. On n'a aucune idée de la complexité (on peut toutefois supposer qu'elle croît avec la longueur). D'autres mesures sont alors nécessaire pour caractériser la taille, comme le nombre de jetons.

2.1.2 Le nombre de jetons (token).

Dans un programme, certaines lignes sont plus difficiles à coder que d'autres. On peut alors leur donner plus de poids. Par exemple une ligne qui référence des pointeurs, un tableau à 2 dimensions et des records aura plus de poids qu'une ligne qui ne fait qu'incrémenter une variable. On peut alors évaluer la taille par le nombre de jetons qui sont des unités de base syntaxique.

C'est ce qu'a introduit Halstead dans "Logiciel Science" [HALS77].
Un programme est composé d'opérateurs et d'opérandes. Un opérateur est un symbole qui spécifie une action; par exemple: Div, +, -, des noms de commande (while, for, read), des parenthèses, des noms de fonctions (EOF). Une opérande est un symbole qui représente une donnée: une variable, une constante, un label.
Les métriques de base sont:

$\eta 1$ = Nombre d'opérateurs uniques.
$\eta 2$ = Nombre d'opérandes uniques.
N1 = Nombre total d'occurrence d'opérateurs.
N2 = Nombre total d'occurrence d'opérandes.

La taille d'un programme en terme du nombre de jetons est: $N = N1 + N2$.

On peut déduire Ss à partir de N grâce à la relation approximative:

$Ss = N \div C$ où C dépend du langage.

Par exemple en fortran, C=7.

De plus Halstead définit des métriques additionnelles comme le vocabulaire η:

$\eta = \eta 1 + \eta 2$,à partir duquel on peut calculer le volume V du programme.

$V = N \log_2(\eta)$. V a pour unité le bit.

Ainsi, si on a $\eta = 27$ et $N = 93$, on a $\log_2(\eta) = 4{,}75$: avec 5 bits, on pourra représenter les 27 différents opérateurs et opérandes du vocabulaire. Comme on a 93 jetons, avec $V = 93*5 = 465$ bits, on pourra stocker tout le programme (non compilé, seulement traduit) en mémoire.

Des études [CHRI80] ont montré que Ss, N et V sont dépendants linéairement et sont des mesures valides de la taille. N et V sont robustes (une variation des règles n'a pas beaucoup d'effet sur la taille).

Comme pour les lignes de code, il n'y a pas d'accord général sur ce qui doit être compté. Halstead ne compte pas les instructions de déclaration et les instructions d'E/S (Read, Write).
Mais actuellement, la tendance est de compter ce genre d'instructions.
Les règles pour compter les jetons ne sont pas standards et dépendent du langage.
Le nombre de jetons permet intuitivement de mesurer plus que la longueur du code. Il donne également une idée de la complexité, indiquant le nombre total de variables et d'opérateurs.

2.1.3 Les mesures de "taille équivalente".

Dans l'industrie, les programmeurs, dans plus de 50 % des cas, modifient des programmes en réutilisant de l'ancien code. Ils peuvent ajouter par exemple du nouveau code. Se pose alors le problème de savoir s'il faut compter les Loc du programme entier ou alors s'il faut pas compter les Loc qui étaient déjà écrites.
La mesure de taille équivalente Se est alors particulièrement intéressante pour évaluer l'effort. L'effort pour développer un logiciel avec Sn lignes de nouveau code et Su lignes de code réutilisé est "équivalent" à l'effort pour développer un logiciel avec Se lignes de code nouvelles. Se est une fonction de Sn et Su.

Boehm, dans son modèle d'estimation de l'effort Cocomo a proposé une fonction:

$Se = Sn + (A \div 100)$ où $A = 0{,}4DM + 0{,}3CM + 0{,}3IM$.

DM est le pourcentage de modifications dans le design, CM est le pourcentage de modification dans le code et IM est l'effort nécessaire pour intégrer le code modifié. La valeur maximale pour A est 100: c'est le cas où il est aussi difficile d'adapter un code existant que de le réécrire complètement.

Bailey et Basili [BAIL81] propose une mesure plus grossière:

$Se = Sn + k$ avec $k = 0{,}2$.
Cette valeur a été trouvé raisonnable, d'après une étude de plusieurs cas.

Thebaut [THEB83] fait l'hypothèse que la contribution du code réutilisé n'est pas linéaire:

$Se = Sn + Su\, e^{k}$ avec $0 \leq k \leq 1$. Par statistique, la constante k est estimé à 6/7.

Il n'y a toujours pas de consensus pour évaluer Se avec ses trois équations qui sont basées sur des données provenant d'un environnement spécifique. On a des résultats qui sont parfois similaires mais qui peuvent être variables. Il est alors délicat d'évaluer Se.

2.1.4 La prédiction de la longueur.

Les métriques précédentes avaient pour but de mesurer la longueur d'un code existant. Mais on a souvent besoin de prédire la longueur tôt dans le cycle de développement de logiciel, en particulier à partir du design. Le nombre de modules ou le nombre de fonctions peuvent permettre de prédire la longueur.

Pour des grands programmes, on peut prédire plus facilement le nombre de modules que le nombre de fonctions, mais même si beaucoup de chercheurs s'accordent pour dire qu'un module devrait avoir environ 100 Loc, on a constaté que la taille d'un module pouvait varier grandement: de 10 à 10000 Loc. Connaissant le nombre de modules, on aura donc du mal à prédire la longueur du code.

On préfère alors utiliser le nombre de fonctions pour prédire le longueur. En effet, le nombre de Loc d'une fonction varie bien moins que pour un module. Ceci est dû au fait qu'une fonction correspond conceptuellement à une tâche d'un programme et comme les capacités mentales humaines sont limitées, la taille d'une fonction est limitée aussi.
Et puis, Basili [BASI79] a montré que pour résoudre un problème, différents programmeurs auront le même nombre de fonctions mais pourront avoir un nombre variable de modules.

On peut également prédire la longueur du code à partir de la longueur du document de design (mesurer souvent en nombre de pages de texte et de diagramme). On utilise pour cela des ratios d'expansion (subjectifs malheureusement). Par exemple, le ratio d'expansion α du design vers le code est égal à:

α = *(taille du design en nbre de page/ taille du code en Loc).*

On prédit le nombre de lignes de code à partir de la formule:

$$LOC = \alpha * \sum_{i=1}^{m} Si\ ,$$

où m est le nombre de modules et Si est la taille du design en nombre de pages pour le module i.

Pour avoir des mesures plus précises, il faut collecter des données propres à son environnement.
Halstead a proposé dans "Logiciel Science" une mesure pour estimer la longueur η d'un programme bien structuré en nombre de jetons.

$$\eta = \eta 1 \ln(\eta 1) + \eta 2 \ln(\eta 2).$$

où $\eta 1$ est le nombre d'opérateurs uniques et
$\eta 2$ est le nombre d'opérandes uniques, comme défini en 2.1.2.
Cette équation se justifie uniquement à partir d'études empiriques et elle reste subjective.

2.1.5 Des métriques de la fonctionnalité.

La fonctionnalité d'un produit est une notion intuitive du nombre et de la complexité de fonctions qui sont délivrées. Nous allons voir ici l'approche de Albrecht qui permet de calculer des "function points" (FP) qui sont des mesures de la fonctionnalité d'un système décrit à partir des spécifications.

On doit pour cela calculer 2 facteurs:

1. UFC (Unadjusted Function Count) est calculé à partir du nombre d'items des types suivants:

- inputs externes: par exemple, ce sont des noms de fichiers et de sélections dans le menu.
- outputs externes: par exemple, ce sont des rapports ou des messages.
- besoins internes: ce sont des inputs interactives qui demandent une réponse et qui ne sont ni des inputs ni des outputs externes.
- fichiers externes: ce sont des interfaces vers d' autres systèmes.
- fichiers internes: ce sont des fichiers logiques du système.

Pour chacun des items identifiés, on attribue un poids en fonction de sa complexité subjective (qui peut être simple, moyenne ou élevée). Le poids est fonction de la complexité et du type d'items.
UFC est égal à la somme des items, affectés de leurs poids.

2. TCF (technical complexity factor) est calculé à partir de 14 facteurs comme la facilité d'installation, la complexité de l'interface, la performance. On note chacun de ces facteurs à l'aide d'une échelle allant de 1 à 5 (5 signifie que le facteur est essentiel, 1 qu'il est insignifiant). TCF varie entre 0,65 et 1,35.

FP est égale à: FP= UFC*TCF.
FP est utilisé en fait dans les modèles de prédictions de la productivité (voir chapitre 4) et mesure plus que la fonctionnalité d'un produit. Les FP, comme on l'a vu incluent des notions subjectives de complexité.

2.2 Les Métriques de Structure Logique.

2.2.1 Les flowgraphes pour modéliser la structure logique.

La structure d'un programme est modélisée par un flowgraphe. Un flowgraphe est composé de noeuds (qui correspondent à une instruction d'un programme) et d'arêtes (qui indiquent le flux de contrôle entre 2 instructions). Les flowgraphes primaires sont souvent rencontrés. Ce sont les suivants:

- Pn (n>0): voir *figure 2.3*.
- D_0: il correspond à l'instruction "if a then A".
- D_1: voir *figure 2.1*.
- D_2: il correspond à l'instruction "while a do A".
- D_3: il correspond à l'instruction "repeat A until a".
- D_4: il correspond à l'instruction "loop(A;B) exit when a".
- D_5: il correspond à l'instruction "if a or else b then X else Y".
- Cn (n>3): voir *figure 2.1*.

Chaque flowgraphe (excepté les flowgraphes primaires) peut être obtenu par une composition de séquence et/ou de nesting de flowgraphes primaires.

- La Séquence: sur la *figure 2.1*, à partir de 2 flowgraphes D_1 et C_3, on produit un nouveau flowgraphe (D_1;C_3). On utilise pour cela le processus de séquence, qui correspond à la notion de concaténation. Sur le flowgraphe (D_1;C_3), le noeud de sortie (représenté par un rond noir) de D_1 et le noeud d'entrée (représenté par un rond noir) de C_3 sont identiques.

- Le Nesting: sur la *figure 2.1*, le flowgraphe D_1(C_3 on B) est obtenu en appliquant le processus de nesting (qui correspond à la notion de substitution). Pour cela, on remplace l'arête partant de B par le flowgraphe C_3.

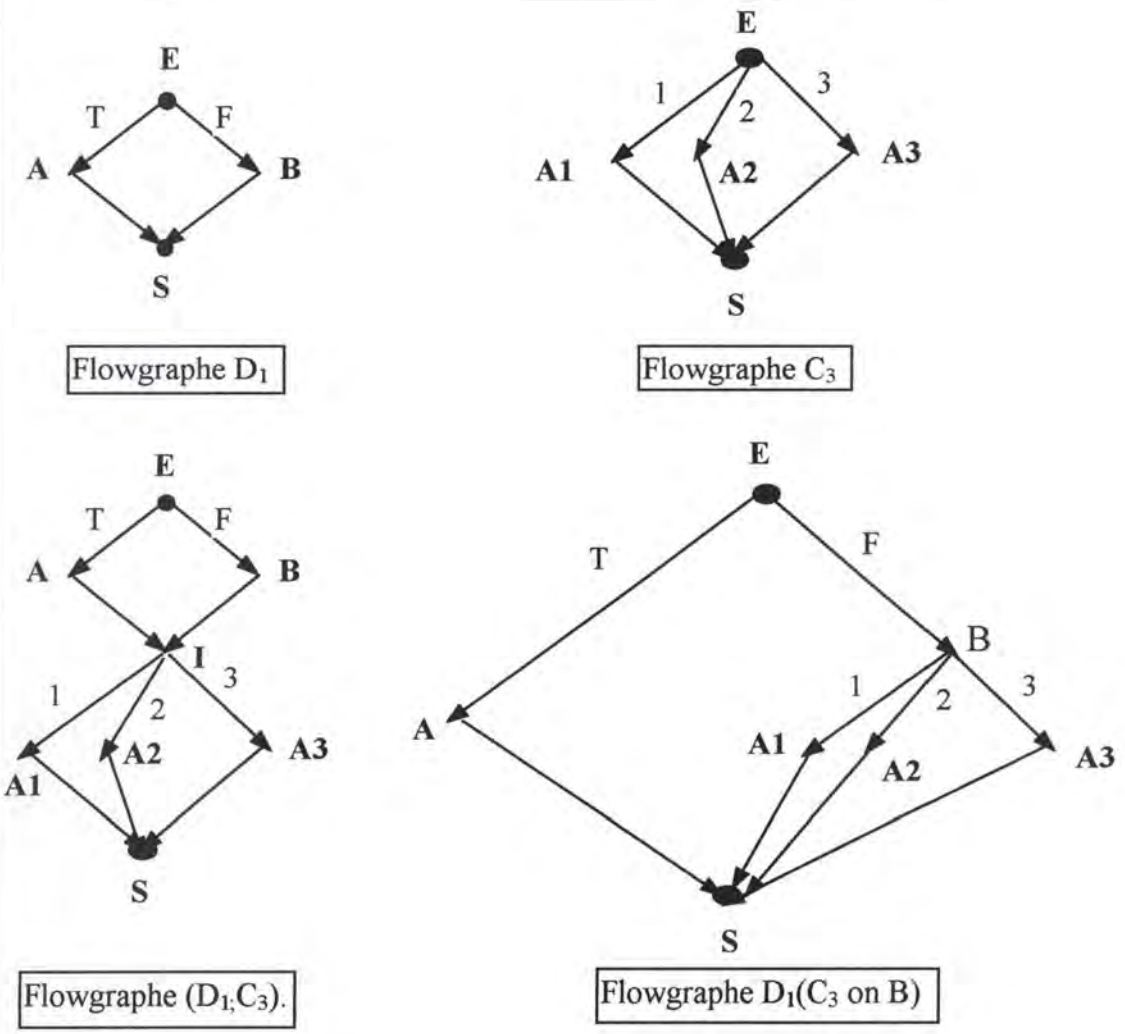


Figure 2.1: Séquence et nesting de flowgraphes.

Ces flowgraphes peuvent être utiles pour voir si la structure du programme est correcte.

Il y a plusieurs définitions de programmation structurée mais les plus populaires disent toutes qu'un programme ne devrait être fait que de séquences, sélections et itérations (les "goto" sont donc exclus).
Pour voir si un programme est bien structuré, on définit d'abord une famille de S-flowgraphes primaires qui correspond aux structures permises selon sa conception de bonne structuration. On décompose ensuite le flowgraphe correspondant au programme en flowgraphes primaires. Cette décomposition est unique. Pour que la structure du programme soit légale, il faut que tous les flowgraphes primaires obtenus par décomposition appartiennent à la famille de S-flowgraphes.

2.2.2 Les niveaux de nesting.

La notion de nesting relative aux flowgraphes a été abordé en 2.2.1 Au niveau du code, le nesting permet au programmeur d'éviter des conditionnelles trop composées mais introduit une complexité conceptuelle. A chaque ligne du code, on peut assigner un niveau de nesting, en prenant pour règles:

1. La première instruction exécutable à un niveau de 1.
2. Si une instruction A a un niveau L et si l'instruction B suit séquentiellement A, alors B a un niveau L aussi.
3. Si A a un niveau L et B est dans une boucle ou une conditionnelle venant de A, alors B a un niveau L+1.

```
                                          niveau
For i :=1 to n-1 do begin                 1
        if x(i) >0 then begin             2
                while x(i)>x(i+1) do begin  3
                        x(i) :=x(i-1)     4
                        end
                end
        end
```

Figure 2.2: Les niveaux de nesting d'un programme.

Une métrique importante est la profondeur du nesting. Plus on va en profondeur, plus le programmeur aura du mal à comprendre quelles conditions doivent être satisfaites pour atteindre une instructions à un niveau élevé. Pour une procédure, on peut définir NL (le niveau de nesting moyen) en assignant à chaque instruction exécutable un niveau de nesting.

2.2.3 Des métriques de complexité logique.

Une mesure simple est la De (decision count) qui représente le nombre de If, Do, While, Case et autres instructions de boucles et conditionnelles. On s'attend qu'un programme avec une valeur élevée de De soit plus compliqué qu'un avec une petite valeur. Si on veut une mesure plus sophistiquée, on peut compter le nombre de prédicats.
Par exemple: « IF c1 and c2 THEN S » contient 2 tests. En fait, cette expression est équivalente à « IF cI THEN IF c2 THEN S ». On a alors 2 prédicats mais un seul IF. Suivant la définition de De, on aura De=1 ou 2.

Une métrique plus connue et plus sophistiquée est le nombre de complexité cyclomatique (V(G)) d'un graphe, proposé par Mac Cabe [MCCA76]. Cette métrique doit donner une idée du nombre de chemins linéairement indépendants dans un programme (c'est important pour savoir s'il sera facile de tester et maintenir un programme. Si V(G) est supérieur à 10, la maintenance sera difficile).

Plus V(G) sera grand, plus il y aura de chemins et plus il faudra faire de tests.

$$V(G) = E - N + 2.$$

où E est le nombre d'arêtes et N est le nombre de noeuds du graphe.
Pour l'exemple de la *figure 2.1*,
$V(G) = 10 - 6 + 2 = 6$ pour le flowgraphe (D_1;C_3).
$V(G) = 9 - 7 + 2 = 4$ pour le flowgraphe D_1(C_3 on B).

Le 2 intervenant dans la formule de V(G) est dû au fait que pour un flowgraphe simple (sans test et sans boucle), comme pour celui de la *figure 2.3*, on doit avoir $V(G) = 1$, vu qu'il n'y a qu'un seul chemin possible.

P_2

$V(G) = 2 - 3 + 2 = 1 = N - 1$

Figure 2.3: La complexité cyclomatique d'un flowgraphe simple.

Si maintenant on a B prédicats dans le graphe,
$E = N - 1 + B$.

D'où $V(G) = E - N + 2 = (N - 1 + B) - N + 2 = B + 1$.
Et si on prend $De = B$, (ce qui est conseillé), on obtient:
$V(G) = De + 1$. Comme la complexité cyclomatique V(G) ne dépend que du nombre de prédicats, ce n'est pas une mesure de structure de contrôle générale.

La mesure Vinap ε, proposée par Boehm, semble être la meilleure mesure de complexité. On a besoin de décomposer le flowgraphe en flowgraphes primaires pour calculer ε.

$\varepsilon(F) = \sum_{n \in nodes} od(n) - 1$ pour un flowgraphe primaire.
Od(n) est le nombre d'arêtes qui partent de *n*.

Séquence: $\varepsilon(F_1,..Fn) = \sum_{i=1}^{n} \varepsilon(Fn)$

Nesting: $\varepsilon(F_1,..,Fn) = \varepsilon(F) * \alpha(F(F_1,..,Fn)) + \sum_{i=1}^{n} \varepsilon(Fi - \varepsilon(P_1)$

où α est la profondeur de nesting du graphe tout entier.

Ces mesures ont l'avantage qu'elles respectent des axiomes que devrait intuitivement respecter des mesures de structure de contrôle générale. Ces axiomes sont les suivants:

Si F, F_1,.., Fn, G,H sont des flowgraphes primaires (différents de P_1), et si $\mu(H) > \mu(G)$ alors:

$F \neq P_1 \Rightarrow \mu(F) > \mu(P_1)$

$\mu(F,G) > \max(\mu(F), \mu(G))$

$\mu(F,G) = \mu(G,F)$

$\mu(F,H) > \mu(F,G)$

$\mu(F(F_1 \text{ on } a)) = \mu(F(F_1 \text{ on } b))$

$\mu(F(H,F_2,..,Fn)) > \mu(F(G,F_2,..Fn))$

$\mu(H(F)) > \mu(G(F))$

$\mu(F(G)) > \mu(F,G)$

$\mu(H(G)) > \mu(G(H))$

2.2.4 Le nombre minimal de chemins.

Scheidewind et Hoffman [SCHN79] ont définit Np comme le nombre minimal de chemins[4] ce qui correspond au nombre de séquences uniques d'arêtes du noeud de début E au noeud de terminaison S.
Ainsi, si on passe 3 fois dans la même boucle, on ne comptera qu'un seul passage.
Np est utilisé pour estimer le nombre de tests à faire lorsque la stratégie du "simple path testing" est appliquée.
Il y a plusieurs stratégies de tests:

- Tester tous les chemins: on doit exécuter chaque chemin une fois au moins. S'il y a une boucle, ce n'est pas toujours possible car on peut avoir un nombre infini de chemins.
- Branch testing: on visite au moins une fois chaque arête.
- Statement testing: on visite au moins une fois chaque noeud.
- Simple path testing: on exécute chaque chemin qui ne contient pas la même arête plus d'une fois. Np est le nombre de chemin à tester quand on applique cette stratégie, qui a été définie par Prather.

Np >= V(G) car en fait chaque fois qu'on ajoute une décision, on ajoute au moins un chemin en plus.

Pour chaque noeud, on peut définir R, qui est le nombre de chemins uniques pour atteindre un noeud.

$\overline{R}$ est le nombre total de chemins divisé par le nombre de noeuds du graphe.

[4] Sous-entendu: "à tester".

Pour la *Figure 2.2*,

$R(E) = R(A) = R(B) = 1$
$R(I) = 2$ chemins $[(E,A,I)\ (E,B,I)]$
$R(A_1) = 2$ chemins $[(E,A,I,A_1)\ (E,B,I,A_1)]$
$R(A_2) = 2$ chemins $[(E,A,I,A_2)\ (E,B,I,A_2)]$
$R(A_3) = 2$ chemins $[(E,A,I,A_3)\ (E,B,I,A_3)]$
$R(S) = 6$ chemins $[(E,A,I,A_1,S)\ (E,B,I,A_1,S)]$
$[(E,A,I,A_2,S)\ (E,B,I,A_2,S)]$
$[(E,A,I,A_3,S)\ (E,B,I,A_3,S)]$

$$\bar{R} = (1+1+1+2+2+2+2+6) \div 8 = 17 \div 8 = 2{,}125$$

2.2.5 L'usage des transferts.

En 2.2.1, la programmation structurée a été abordée. Un problème important est l'usage des transferts (c'est à dire des "goto"). Dijkstra suggère qu'un programmeur avisé doit éviter d'utiliser les "goto" car cela rend la compréhension du programme plus difficile: il est bien mieux que des instructions successives dans le programme correspondent à des actions successives dans le temps.
Dijkstra ajoute qu'il a observé que la qualité des programmes est une fonction décroissante du nombre de "goto" dans un programme. D'où l'importance de compter le nombre de transferts directs. Mais il faut remarquer que pour certains langages comme le fortran, le "goto" est indispensable et cette mesure perd de son sens. De même, l'usage des transferts est parfois nécessaire pour éviter de dupliquer du code inutilement.

Une mesure plus fine est le nombre de croisements, proposée par WoodWard, Hennell et Hedley [WOOD79]. C'est une métrique du nombre de "goto" non contrôlé. Ce nombre doit être le plus bas possible.

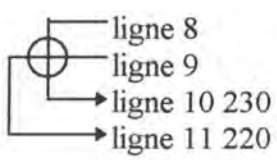


Figure 2.4: un exemple de "croisement".

Soit (A,B) une paire signifiant qu'on a un transfert de A vers B.
On a un croisement (voir *figure 2.4*) si, considérant 2 paires (A,B) et (C,D) un des 2 cas suivants est vrai:

- $\min(A,B) < \min(C,D) < \max(A,B)$ et $\max(C,D) > \max(A,B)$.
- $\min(A,B) < \max(C,D) < \max(A,B)$ et $\min(C,D) < \min(A,B)$.

On doit essayer le plus possible de réarranger le programme pour qu'il y ait le moins de croisements possibles.

2.3 Les Métriques de Structure de Données.

Il est important d'avoir des mesures de structure de données si on veut avoir une vue générale de la complexité d'un système. Mesurer la complexité de la structure logique ne suffit pas car la complexité peut être cachée dans la structure de données. 2 programmes peuvent être équivalents dans le sens où ils ont la même fonction mais ils peuvent avoir des structures de données différentes.

2.3.1 La quantité de données.

On peut mesurer la quantité de données utilisées à partir du nombre total de variables: Var.
Halstead introduit en outre $\eta 2$ (le nombre d'opérandes uniques) et N2 (le nombre total d'occurrence d'opérandes).

Var, $\eta 2$ et N2 sont les métriques les plus populaires de structure de données. Le problème est qu'ils ne tiennent pas compte de la différence entre les types de données simple (comme les entiers, les caractères ou les booléens) et les types de données complexes (comme les tableaux, les records). Il serait alors intéressant d'assigner un poids aux variables pour donner une idée de la complexité de la structure de données.
Les métriques qui suivent sont moins utilisées.

2.3.2 L'usage des données dans un module.

Pour caractériser l'usage des données dans un module, 2 métriques sont utilisés: les variables actives (live) et l'étendue des variables.

2.3.2.1 Les variables actives.

Quand on programme, on doit garder en tête le status de certaines variables: par exemple, dans une boucle, on doit se demander quelle valeur aura la variable qui sert de test pour la condition de sortie. Cette variable est alors active (live). On a plusieurs définitions de variable active mais la plus courante est:
« Une variable est active entre sa première et dernière référence, au sein d'une procédure. »
Pour chaque ligne, on peut compter le nombre de variables active.
LV est la moyenne du nombre de variables active pour une procédure. C'est la somme du nombre de variables active pour chaque ligne divisée par le nombres d'instructions exécutables dans une procédure.

2.3.2.2 L'étendue des variables.

Span (A) est une métrique qui mesure le nombre d'instructions entre 2 références successives à la variable A.
Par exemple, si on considère le programme de la *figure 2.5*, la variable A a 2 étendues de 19 et 9 lignes. On peut alors faire une moyenne de la taille de l'étendue de A. Si une variable A a de larges étendues, cela demandera au programmeur un effort intellectuel pour se rappeler longtemps de cette variable alors qu'on ne la référence pas souvent.

```
ligne 01        X :=A ;
...
ligne 20        Y :=A ;
...
ligne 30        Z :=A ;
```

Figure 2.5: les références d'une variable.

2.4 Les Métriques de Modularité.

Un module est une ensemble d'instructions de programme qui est vu comme un concept simple. Un module doit être théoriquement compilable et testable séparément. La modularisation permet de réduire la complexité, de faciliter la gestion mais implique des besoins d'interfaçage et de coordination entre les modules.
Il est important d'avoir une bonne découpe en module, car c'est un signe de bon design.

2.4.1 Les modèles de modularité.

Les charts de design (voir *figure 2.6*) décrivent le flux d'information passé entre les modules, c'est à dire les variables qui sont passées entre les modules.

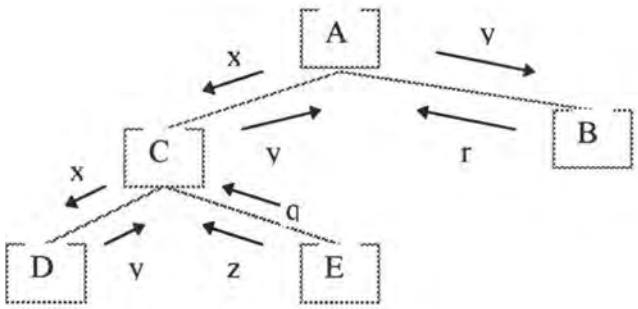
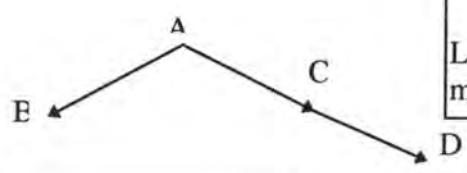


Figure 2.6: Un chart de design.

Les callgraphes permettent de voir quel module appelle quel module (voir *figure 2.7*).



A appelle B et C.
C appelle D.

Le noeud racine A correspond au module de plus haut niveau

Profondeur: 2
Largeur: 2
Nombre de noeuds: 4
Nombre d'arêtes: 3
Ratio nbre noeuds/nbre arêtes: 1.33

Figure 2.7: Un callgraphe.

Des métriques comme:

M_1 = *nombre de Modules / nombre de Procédures.*

M_2 = *nombre de Modules / nombre de Variables.*

donnent une idée de la modularité globale du système.
Mais la morphologie du système est plus intéressante.

2.4.2 La morphologie.

Yourdon et Constantine utilise la notion de morphologie pour référencer la forme de la structure globale du système.

Des mesures comme celles-ci dessous donnent une indication de la qualité du design:

- la profondeur (le chemin le plus long du noeud racine à un noeud feuille où un noeud représente un module).
- la largeur (le nombre maximal de noeud à un certain niveau).
- le nombre de noeuds.
- le nombre d'arêtes.
- le ratio nombre de noeuds/nombre d'arêtes (ce ratio donne une mesure de la densité de connectivité. Plus ce ratio est élevé, plus il y a de connexion entre les noeuds).

Un exemple d'application de ces mesures est présenté avec la *figure 2.7*.

Pour avoir un bon design, le callgraphe doit être le plus proche possible d'un arbre (un arbre est composé de n noeuds et n-1 arêtes). Ince et Hekmat énonce:
"Plus la structure du système est éloigné d'un arbre, plus le design est mauvais."

Un callgraphe G qui n'est pas un arbre est dit "impur". Il y a dans sa structure un ou plusieurs spanning trees; un spanning tree étant un arbre qui a le même nombre de noeuds que le callgraphe G impur.
Exemple: Le graphe G_2 de la *figure 2.8* est impur. Il y a dans sa structure 2 spanning trees:
- Le graphe G_2 avec l'arête 1 en moins.
- Le graphe G_2 avec l'arête 2 en moins.

Une mesure m(G) de l'impureté d'un callgraphe G doit avoir les propriétés suivantes:
1. $m(G) = 0$ si et seulement si G est un arbre.
2. $m(G_1) > m(G_2)$ si G_1 a plus d'arêtes que G_2.
3. Si G_1 et G_2 ont une arête en plus que leur spanning tree et si G_1 a moins de noeuds que G_2 alors $m(G_1) > m(G_2)$.

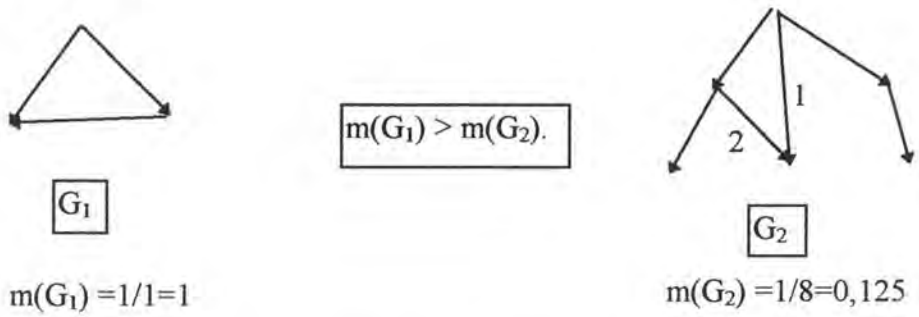


Figure 2.8: L'impureté des callgraphes.

4. pour chaque callgraphe G, $m(G) \leq m(K_n) = 1$ où N est le nombre de noeuds de G et K_n est un callgraphe de N noeuds

Une mesure qui satisfait ces 4 propriétés est la suivante (exemple à la *figure 2.8*):

$$m(G) = \frac{\text{nombre d' aretes en plus que le spanning tree}}{\text{nombre maximal d' aretes en plus que le spanning tree}}$$

Dans un graphe complet, $e = \frac{n(n-1)}{2}$ où e est le nombre d'arêtes et n est le nombre de noeuds.

Un arbre a n-1 arêtes. D'où le nombre d'arêtes en plus que le spanning tree est:

$$\frac{n(n-1)}{2} - (n-1) = \frac{(n-1)(n-2)}{2}$$

Le nombre d'arêtes en plus que le spanning tree est: $e-(n-1) = e-n+1$.

D'où: $\boxed{m(G) = \frac{2(e-n+1)}{(n-1)(n-2)}}$

m(G) doit être le plus proche possible de 0.

2.4.3 Le couplage.

Le couplage est une mesure du degré d'interdépendance entre les modules. C'est un attribut qui caractérise une paire de module. Troy et Zweben énonce:
" Le couplage est un des attributs le plus important affectant la qualité du design".

Il y a 5 relations possibles de couplage entre un module x et un module y (de la pire à la meilleure):

- R_5: couplage sur le contenu: x réfère l'intérieur de y, c'est à dire effectue un branchement dans y ou change des données d'une instruction de y.
- R_4: couplage commun: x et y réfère la même donnée globale. Ce genre de couplage est indésirable car si le format de la donnée globale doit être changé, il faudra changer dans tous les modules qui réfèrent cette donnée.
- R_3: couplage de contrôle: x passe un paramètre à y pour le contrôler (par exemple un drapeau).
- R_2: couplage stamp: x et y accepte le même type d'enregistrement comme paramètre.
- R_1: couplage de données: x et y communique par paramètre.
- R_0: pas de relation: x et y sont totalement indépendants.

L'idéal est de n'avoir que des relations R_0.

2.4.4 La cohésion.

La cohésion est un attribut qui décrit la solidité relative fonctionnel d'un module.
Il y a 7 types de cohésion (de la meilleure à la pire):

- La cohésion fonctionnelle: le module exécute une fonction bien définie.
- La cohésion séquentielle: le module exécute plus d'une fonction mais dans un ordre décrit dans les spécifications.
- La cohésion communicationnelle: le module exécute plus d'une fonction mais elles ont toutes le même ensemble de données.
- La cohésion procédurale: le module exécute plus d'une fonction mais elles sont toutes relatives à la même procédure générale du logiciel.
- La cohésion temporelle: le module exécute plus d'une fonction mais elles sont dans le même intervalle de temps.
- La cohésion logique: le module exécute plus d'une fonction mais elles sont reliées logiquement.
- Pas de cohésion: le module a été établi par coïncidence.

Un module peut avoir plusieurs types de cohésion.

Le ratio cohésion permet de mesurer la cohésion globale du système:

$$\textit{ratio}\ \text{cohésion} = \frac{\text{nombre de modules avec une cohésion fonctionnelle}}{\text{nombre total de modules}}$$

2.4.5. Le flux d'information.

Henry et Kafura définissent le Fan-in et le Fan-out d'un module pour mesurer le flux d'information entre les modules.
2 notions sont à introduire pour définir le Fan-in et le Fan-out:

- Le flux local direct d'un module M: le module M invoque un autre module et lui passe de l'information ou le module M est invoqué par un autre module et lui retourne un résultat.
- Le flux local indirect d'un module M: un module M passe de l'information à un module M' qui transmet la même information à un module M".

Le Fan-in d'un module M est le nombre de flux locaux qui finissent en M plus le nombre de structures de données retrouvées par M.
Le Fan-out d'un module M est le nombre de flux locaux qui émanent de M plus le nombre de structures de données mises à jour par M.

Le Fan-in et le Fan-out permettent de mesurer la complexité d'un module M:

$$\text{Complexité d'un module M} = \text{Longueur (M)} * (\text{Fan-in(M)}*\text{Fan-out(M)})^2.$$

D'après cette mesure, les modules isolés du système auront un Fan-in et un Fan-out faibles et seront peu complexes. Ceci correspond à la notion intuitive de la complexité car plus un module reçoit et transmet de l'information, plus il est complexe. Néanmoins, il est douteux qu'un module avec un Fan-in ou un Fan-out nul ait une complexité nulle, comme l'indique cette métrique.

Les métriques qui suivent sont des mesures d'attributs externes et sont utiles pour les modèles qui sont présentés au chapitre 4.

2.5 Les Métriques relatives à l'Effort et au Coût.

Pour un manager, il est important de connaître l'effort et le coût d'un projet pour savoir s'il est faisable et profitable. On distingue 2 catégories de modèles et de mesure: le niveau micro pour les petits projets, avec des programmeurs individuels et le niveau macro pour les grands projets avec des "équipes" de programmeur.

2.5.1 Le niveau micro d'effort et de coût.

L'effort se mesure en heures/personnes. On enregistre donc le temps passé pendant le développement pour évaluer l'effort. Mais se pose le problème de savoir ce que l'on doit compter exactement: doit-on prendre en compte les interruptions? Les résultats ne sont alors pas toujours transportables d'un environnement à un autre, chaque organisation ayant sa spécificité pour évaluer l'effort. Mais il est établit de ne pas enregistrer le temps passé pour comprendre un

problème nouveau ou pour s'habituer à un nouveau langage car en fait ceci doit faire partie de l'éducation. Connaissant le coût d'un programmeur à l'heure et le nombre d'heures, il est possible d'évaluer le coût du programme, en prenant aussi en compte les coûts en plus (overhead costs) comme les frais de management, de secrétariat, etc.

Il est possible de calculer l'effort par d'autres métriques que le temps comme le nombre de fois qu'un programme est lancé. Mais ceci est moins fiable car certains programmeurs modifient peu leurs programmes avant de les relancer alors que d'autres passent des heures avant de les relancer.

2.5.2 Le niveau macro d'effort et coût.

On mesure ici les métriques d'effort en mois/personne ou années/personne. La différence entre le niveau micro et macro est au niveau de l'intensité: on ne peut s'attendre à ce que l'intensité en programmant pour un petit projet persiste pendant plusieurs mois. On ne peut alors comparer les résultats de productivité entre le niveau micro et macro. Le fait qu'un programmeur, sur une journée de 8 heures, ne passent que 4 à 5 heures à programmer, étant le reste du temps en meeting ou en pause [PUTN78] n'aura peut être pas d'effet au niveau micro mais au niveau macro, il faudra en tenir compte.
Souvent on remplit des formulaires pour évaluer l'effort mais au niveau macro, c'est parfois au programmeur de décider combien de temps il a passé sur un problème.

Il est plus difficile de calculer le coût au niveau macro. Il ne suffit pas de multiplier le temps estimé par le salaire moyen par unité de temps mais il faut tenir compte du coût des voyages, de formation, de dépenses administratives, etc, qui dépendent de l'environnement.

2.6 Les Métriques de Reliability.

Malgré les efforts dépensés pour corriger et tester un logiciel, il n'est pas possible de produire du code sans erreur avec une garantie totale. Une faute est une erreur dans le logiciel qui fait que le logiciel produit un résultat incorrect avec une entrée valide.
Un logiciel peut mal se comporter pour différentes entrées à cause de la même faute et une entrée peut révéler plusieurs fautes.

2.6.1 Les métriques de fautes.

Il y a 3 métriques typiques pour évaluer les fautes dans un logiciel:

- Le nombre de changements requis dans le design: cette métrique reflète la mauvaise compréhension des spécifications. Les changements peuvent avoir lieu n'importe quand dans le cycle de vie du logiciel à partir de la phase de design.

- Le nombre d'erreurs dans le codage: dans beaucoup de cas, lorsque l'on découvre une erreur (souvent pendant la phase de test), on remplit un rapport. Le nombre de rapports peut servir de métrique.
- Le nombre de changements dans le programme: en effet, beaucoup de fautes sont corrigées par des changements de code. Bien évidemment, on doit éviter de compter les changements dus à une modification des spécifications ou l'ajout de lignes de commentaires. Ces 2 dernières mesures dépendent de la qualité des tests. Une valeur faible de faute peut provenir d'un bon design et codage mais aussi de mauvais tests qui ne découvrent pas les fautes.

Une faute peut demander plusieurs changements. On peut alors caractériser la complexité d'une faute par le nombre de changements qu'elle a nécessité.
Le problème est que si on rajoute 10 ou 500 lignes, on ne comptera qu'un changement.
On peut comparer 2 programmes à partir de leur densité de fautes qui est le nombre de fautes divisé par le nombre de lignes du programme.
Pour caractériser la sévérité d'une faute, on peut compter le nombre de lignes changées.

Remarquons que la difficulté à corriger une faute est indépendante de sa sévérité. Il est alors intéressant de mesurer le temps requis pour corriger une faute.

3.6.2 Découverte et correction des fautes.

Une faute peut être découverte pendant la maintenance et provenir d'une mauvaise compréhension des spécifications. Avec les méthodes modernes de développement, grâce aux reviews[5] et inspections durant le design et le codage, on peut retrouver plus de fautes plus tôt dans le cycle de développement, ce qui entraîne moins de dépense en correction comme le montre la *figure 2.9*, dont les ratios ont été établit, suite à une expérience chez I.B.M.

Faute découverte:	Ratios
Pendant le **design** ou le **codage**	**01**
Pendant les **tests**	**20**
Par le **client**	**80**

Figure 2.9: Ratios des coûts de correction d'une faute suivant la phase où elle est découverte.

Il est alors intéressant de connaître le nombre de faute découvertes suivant les phases du développement. Ceci est reflété par les métriques suivantes:

- D_0 est le nombre de fautes découvertes pendant le design et le codage. Cette mesure aura un sens si on développe le logiciel de manière structurée, en essayant de ne pas avoir de fautes

[5] ce terme est défini en 6.3.

après le codage et en ne comptant pas sur les tests pour détecter des fautes que l'on aurait pu découvrir avant.

- D_1 est le nombre de fautes pendant la phase de test.
- D_2 est le nombre de fautes découvertes après que le produit soit livré et durant la maintenance. C'est souvent ici que l'on découvre beaucoup de fautes.

3.6.4 Le calcul de la reliability.

Au chapitre 1, une définition formelle a été donnée de la reliability.
Une mesure de la reliability doit donner une idée du nombre d'erreurs contenues dans le code.
La théorie des probabilités est utilisée pour prédire quand une faute sera découverte dans le domaine du hardware. De même, pour les logiciels, il est possible de calculer la probabilité qu'une erreur soit découverte au temps to (notée F(to), to>=0), en faisant l'hypothèse que la détection de fautes est un processus stochastique.

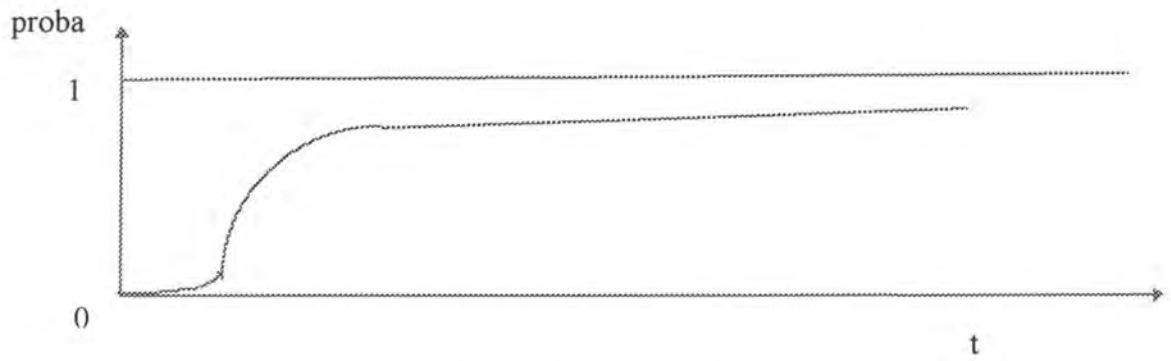


Figure 2.10: Probabilité qu'une faute soit découverte.

La reliability R(to) du logiciel est la probabilité qu'il n'y ait pas de faute durant l'intervalle de temps de longueur to.
R(to) = 1-F(to), en supposant que le logiciel soit utilisé continuellement.
On voit alors l'importance de noter quand les fautes surviennent.

CHAPITRE 3:

COMMENT MESURER LA QUALITÉ.

Un des objectifs principaux du software engineering est d'améliorer la qualité des logiciels. Mais la qualité, comme la beauté, fait l'objet de débats philosophiques. La part de subjectivité est importante dans la définition de ce terme.
On va s'efforcer dans ce chapitre d'identifier les attributs intéressants des logiciels au niveau de l'utilisateur.
Les attributs externes sont dépendants d'entités additionnelles du produit en question. Par exemple, si le produit est le code du logiciel, sa maintenabilité est un attribut externe, étant dépendant aussi du programmeur. La qualité est perçue en terme d'attributs pertinents pour des types particuliers d'utilisateurs. Les attributs internes définis au chapitre 2 ont un impact sur la qualité dans le sens où ils affectent les attributs externes.

On va s'intéresser dans ce chapitre aux attributs externes particulièrement intéressants: ce sont la reliability (discutée en 3.2), l'utilité (3.3) et la maintenabilité (3.4).

Mais voyons d'abord (3.1) comment mesurer les attributs importants de la qualité[6] en se basant sur les différents modèles proposés dans la littérature.

3.1 Des Mesures de la Qualité des Logiciels.

Au chapitre 1, un modèle général de qualité a été présenté. Les attributs clés sont normalement des attributs externes de haut niveau comme la reliability, la maintenabilité mais il y a aussi des attributs internes comme la testabilité ou l'efficacité. Ces attributs de haut niveau ne peuvent pas être mesurés directement. C'est pourquoi on a fait une décomposition[7].

Avant de mesurer les attributs de la qualité que l'on juge importants, il faut pouvoir les déterminer. Une approche générale (définie par Mac Call [MCCA77]) pourrait être la suivante:

[6] définis au chapitre 1.
[7] voir le chapitre 1 pour de plus amples détails.

1. Identifier et assigner une importance relative aux attributs.

2. Comparer le coût pour avoir un attribut avec ce que l'on devra payer si on n'a pas cet attribut. On peut alors revoir son importance. Par exemple, si les coûts pour mesurer et obtenir une système facile à maintenir sont plus élevés que les coûts prévus de maintenance, on ne fera pas d'effort pour avoir un système avec une maintenabilité élevée.

3. Déterminer les relations et conflits entre attributs.

4. Définir les attributs critiques et voir comment développer le logiciel pour obtenir la qualité attendue.

5. Détailler la description des attributs. Par exemple, pour la portabilité, définir sur quel environnement le système pourrait être transporté. On doit aussi spécifier un taux souhaité pour chaque attribut. Par exemple, un taux de maintenabilité pourrait être défini comme suit: "la moyenne pour résoudre un problème doit prendre moins de 5 journées/ hommes".

6. Identifier les métriques à appliquer afin d'avoir une indication de la progression des attributs durant le développement.

Il y a 2 approches particulières pour contrôler la qualité des logiciels:

1. L'approche du modèle fixe: à partir d'un modèle donné, on sélectionne les facteurs[8] de qualité que l'on considère importants et on accepte les critères et métriques associés à ces facteurs comme pertinents pour son organisation particulière.

Par exemple, Mac Call [MCCA77] a développé 41 métriques pour mesurer 23 critères[9] de qualité. Voyons comment Mac Call propose de mesurer le critère "complétude" pour estimer le facteur correction (correctness).
On dispose d'une liste (représentée sur la *figure 3.1*) composée de 9 articles. A chaque article, il faut associer une ou plusieurs phases parmi les suivantes: besoins (B), design (D) , implémentation (I). Il faut répondre par oui ou non pour chacune des phases associées à chaque article (selon qu'à la phase en question la déclaration de l'article est respectée ou non).

[8] ou caractéristiques générales.
[9] ou caractéristiques primitives.

1. Références non ambiguës (input, fonction, output) - B,D,I -
2. Toutes les références de données sont définies, calculées ou obtenues à partir d'une source externe - B,D,I -
3. Toutes les fonctions définies sont utilisées - B,D,I -
4. Toutes les fonctions référencées sont définies - B,D,I -
5. Toutes les conditions et processus sont définis pour chaque point de décision - B,D,I -
6. Lors de appels de procédures, les paramètres sont correctement définies et référencées - D,I -
7. Tous les problèmes reportés sont résolus -B,D,I -
8. Le design a été fait en accord avec les besoins - D -
9. Le code a été développé en accord avec le design - I -

Figure 3.1: Une liste pour mesurer la complétude.

La métrique x pour mesurer la complétude est la suivante:

$$x = 1/3 * [\frac{(\text{nombre de oui pour B})}{6} + \frac{(\text{nombre de oui pour D})}{8} + \frac{(\text{nombre de oui pour I})}{8}]$$

x est compris entre 0 et 1 puisque pour chaque phase on divise par le nombre de oui maximum que l'on peut obtenir.

Dans le modèle proposé par Mac Call, la correction dépend de la complétude, de la traçabilité et de la consistance. La correction C est définie par:

$$C = \frac{(x+y+z)}{3}$$

où y et z sont des métriques pour la traçabilité et la consistance.

2. L'approche "je définis mon propre modèle" (d'abord développée par Gilb): en se basant sur les modèles de qualité, "l'ingénieur de logiciel" et l'utilisateur décident ensemble des attributs importants et de la décomposition (en attributs de plus bas niveau) ainsi que des métriques à utiliser. Ils définissent ensuite des objectifs mesurables facilement vérifiables au fur et à mesure de la livraison incrémentale du produit. Cette technique simple s'avère puissante.

La reliability est l'attribut que l'utilisateur considère généralement comme le plus important. Il s'avère que c'est l'attribut que l'on peut mesurer et prédire le plus facilement.
L'utilité est aussi de grande importance pour l'utilisateur ainsi que la maintenance (qui intéresse aussi le concepteur), vu les coûts excessifs qu'elle engendrent.

Pour certains systèmes, la portabilité peut être importante. La responsabilité reste à l'utilisateur et au concepteur pour mettre en place des objectifs mesurables. Des formules simples comme celle-ci peuvent être utilisées:

Portabilité = 1- (ET/ER)
où ET correspond aux ressources dont on a besoin pour déplacer le système vers son nouvel environnement.
ER correspond aux ressources dont on a besoin pour créer le système dans son environnement actuel.

La plupart des autres mesures des caractéristiques de qualité sont subjectives. Il faut alors les utiliser avec parcimonie.

L'utilisation de ces modèles nécessite la définition d'un plan et la collection de données. Cela demande du temps et les managers sont souvent réticents.
Ils utilisent alors souvent des mesures plus grossières de la qualité comme le taux d'erreur:

Qualité = Nombre de fautes connues/taille.

En l'absence d'autres mesures, celle-ci peut être très utile.
Seulement il faut remarquer que les erreurs ne conduisent pas en général à des problèmes de qualité.
En effet, Adams de IBM [ADAM84] a conclut que:
- 1/3 des fautes détectées apparaissent peut souvent.
- peu de fautes (environ 2%) apparaissent très souvent.
Un produit peut alors avoir beaucoup de fautes et ne provoquer des incidents que rarement.

3.2 La Reliability.

Nous allons nous intéresser dans cette partie à l'estimation et la prédiction de la reliability. Il n'y a pas de technique unique pour cela. On ne discutera pas ici de la mesure de la reliability ultra-élevée, comme le nombre d'heures de vol sans incident d'un Airbus.

Les logiciels ne se comportent pas comme prévu car ils contiennent des erreurs. Lorsqu'elles sont découvertes, on s'efforce de les corriger[10] et la reliability tend alors à augmenter.
Remarquons qu'il n'en est pas de même pour la théorie de la reliability du hardware ou la reliability du système dépend de la reliability des composants. Quand une panne survient, le composant en question est remplacé et le système retrouve la même reliability que celle d'avant la panne. Pour le hardware, on ne intéresse pas aux fautes de design mais aux pannes physiques.

Durant les 25 dernières années, il y a eu beaucoup de recherche pour mesurer la reliability des logiciels. Dans ces études, on suppose que le logiciel fonctionne dans un environnement réel ou simulé et que l'on cherche à corriger les fautes quand des incidents surviennent.

[10] parfois ce n'est pas couronné de succès ou alors d'autres erreurs sont introduites.

Musa a montré que les périodes de temps sans incident sont de plus en plus longues au fur et à mesure que l'on découvre des fautes. Ces périodes peuvent être très courtes ou très longues (de 0 à 6000 secondes) durant les tests.

Les temps d'exécution sans incident sont aléatoires. Comme les inputs qui vont être introduits ne sont pas connus, il est impossible de mesurer exactement l'instant du prochain incident. Le temps d'exécution sans incident est alors décrit par une variable aléatoire et on recourt aux probabilités et aux statistiques pour lui donner une valeur estimée.

La fonction de reliability Ri(t) est définit ainsi[11]:

Ri(t) = P(Ti>t)= 1-Fi(t) où
Ti représente le temps,
P(Ti>t) est la probabilité que le temps d'exécution jusqu'au prochain incident soit supérieur à t,
Fi(t) est la fonction de répartition de Ti.

La section 3.2.1 abordera les prédictions de la reliability. En 3.2.2, les modèles de reliability seront présentés et en 3.2.3, nous verrons que la méthode de recalibration permet d'améliorer les prédictions.

3.2.1 La précision de la prédiction.

Il y a de grandes variations de précision entre les différentes méthodes pour prédire la reliability des logiciels. Il n'y a pas de méthode unique pour donner des résultats précis et la précision varie d'un ensemble de données à un autre. Une méthode peut donner de bonnes prédictions pour certaines données et être très mauvaise pour d'autres.

Pour un même programme, les différentes méthodes ne donnent pas les mêmes prédictions, certains modèles étant très optimistes et d'autres très pessimistes.

L'imprécision peut être due au biais ou au bruit.
Quand les prédictions sont biaisées, au départ on a une estimation raisonnable puis on s'écarte de plus en plus de la valeur réelle de la reliability soit au dessus, soit en dessous.
Quand il y a du bruit, l'estimation fluctue autour de la valeur réelle au dessus et en dessous mais dans à peu prêt les mêmes proportions au cours du temps.

Pour détecter le biais, on peut compter le nombre de fois que la valeur estimée est plus grande (ou plus petite) que la valeur réelle. On cherche en fait à voir la différence entre $\hat{Fi}(t)$ (l'estimation de la fonction de répartition) et Fi(t).

[11] En 3.6.4 la reliability a déjà été présentée.

3.2.2 Les modèles paramétriques de la croissance de la reliability.

Si on fait l'hypothèse que les inputs qui produisent des incidents sont rencontrés aléatoirement, le TTNF (time to next failure) a une distribution exponentielle, de même que les temps inter-incidents.
Si T_1, T_2, etc sont les temps inter-incidents successifs, on a une description complète du processus stochastique si on connaît les taux λ_1, λ_2, etc.

Il y a 2 sortes d'incertitudes: on ne sait pas si une tentative pour corriger une faute a été couronnée de succès et même si c'est le cas, on ne connaît pas l'amélioration de la reliability (a-t-on corrigé une faute majeure ou mineure?).

Les modèles les plus connus sont probablement les premiers.

Le modèle Jelinski-Moranda, une extension du modèle Musa est un des plus utilisés. Dans ce modèle, on suppose qu'un programme contient N fautes, que chaque faute a la même importance ϕ et que les fautes découvertes sont corrigées parfaitement. Les variables indépendantes T_1, T_2 sont distribuées exponentiellement et ont pour paramètres $N\phi$, $(N-1)\ \phi$, etc. C'est à dire:

$$Fi(t) = 1 - e^{-(N-i+1)\phi t}.$$

Un point faible de ce modèle est que les fautes ont toutes la même importance et la séquence des taux est déterminée. Ce modèle conduit généralement à des estimations trop optimistes.

Le modèle Littlewood assignent aux fautes une importance à l'aide de variables aléatoires indépendantes suivant une distribution Gamma de paramètres a et b.
Dans ce modèle, il y a une tendance à corriger les fautes importantes plus tôt que les fautes mineures. Pour le modèle Littlewood, les temps Xj auxquels les incidents sont découverts suivent une distribution de Pareto:

$P(Xj < x) = 1 - [\beta \div (\beta + x)]e^{\alpha}$. (la probabilité décroît exponentiellement quand x croît).

Ces modèles font partie d'une classe générale appelée modèles statistiques d'ordre exponentiel. Dans de tels modèles, les fautes restantes sont découvertes en fonction de leur importance représentée par un taux.
Comme les modèles peuvent être imprécis, on a alors recours à la recalibration.

3.2.3 La recalibration.

Si on arrive à connaître assez tôt l'imprécision des prédictions, on peut alors améliorer les prédictions à l'aide de la recalibration.
Soit $\hat{Fi}(t)$ une prédiction de la fonction de répartition dont Fi(t) est la distribution réelle.

Gi met en relation Fi(t) et $\hat{F}i(t)$:

$$Fi(t) = Gi[\hat{F}i(t)].$$

Si on connaît Gi, on connaît Fi(t).
Dans beaucoup de cas, Gi est pratiquement stationnaire. On peut alors approcher Gi (en faisant une série de prédictions puis en comparant avec la réalité) et avoir une autre prédiction:
$\hat{F}'i(t) = Gi[\hat{F}i(t)]$.

Cette technique de recalibration marche très bien.

3.3 L'Utilité (usability).

Une notion intuitive de l'usability est convivialité (user-friendly)[12]. Il n'y a pas vraiment de mesure communément acceptée de cet attribut.

Une mesure pourrait être: "la probabilité qu'un utilisateur du système ne rencontre pas de problème avec l'interface pendant une période donnée pour un type d'opérations."
Le problème avec cette mesure est qu'elle demande de collecter beaucoup trop d'informations.
Il est communément accepté que des manuels bien structurés, des messages d'erreurs instructifs, des fonctions d'aide et de bonnes interfaces sont des moyens d'atteindre une bonne utilité.
Les attributs internes qui influencent l'utilité sont, hormis le nombre d'écrans d'aide et de menus d'option, des mesures de la structure du texte comme la compréhensibilité ou la lisibilité.

Gilb a décomposé l'utilité en 3 attributs mesurables:

- Le niveau de l'utilisateur (entry level): il peut se mesurer en années d'expérience avec une classe d'applications similaires.

- La facilité à apprendre: elle peut être mesurée par la vitesse d'apprentissage (par exemple en heures d'apprentissage nécessaires pour utiliser le système de manière indépendante).

- La capacité de maniement: on peut la mesurer par la vitesse de travail d'un utilisateur entraîné ou encore par le nombre d'erreurs commises en travaillant à une vitesse normale.

Cette vue est incomplète car elle ne définit pas de mesures objectives et elle ne tient pas compte de la satisfaction de l'utilisateur.

[12]voir le chapitre 1 pour une définition plus formelle

3.4 La Maintenabilité.

Avant de discuter des mesures, voyons quelques caractéristiques de la maintenance. Des mesures seront ensuite présentées selon les vues externes et internes de la maintenabilité.

Par nature, les logiciels changent et évoluent fréquemment. La qualité des anciens produits (développés il y a plusieurs années) est souvent faible. Pour des raisons économiques, les logiciels obsolètes ne sont pas complètement remplacés et la maintenance est là pour assurer les nouveaux besoins. Elle continuera à représenter un grand investissement. Les managers, dont l'objectif ultime est la satisfaction du client, cherchent alors à minimiser les fautes, l'effort et la durée de la maintenance.
La maintenance peut aussi s'appliquer à d'autres produits comme les documents relatifs aux besoins ou au design et même aux documents de planification des tests.

Il y a 3 types d'activités de maintenance (qui ont toutes pour but de faire des changements spécifiques au produit):

- la maintenance corrective: cela consiste à trouver et corriger des bugs. (1/3 des bugs sont dus à une mauvaise compréhension des besoins de l'utilisateur. 60% des fautes sont dues à une logique incorrecte, 5% à des problèmes d'interface, 5% à des problèmes de documentation, 20% à un manque de logique et 10% à une mauvaise correction).

- la maintenance adaptative: cela consiste à modifier le logiciel pour l'adapter au nouvel environnement.

- la maintenance pour améliorer: cela consiste à ajouter une nouvelle fonctionnalité.

On remarque que les fautes sont souvent concentrées dans un petit nombre de modules qui causent problème à cause de leur complexité.

Une mesure de la maintenabilité pourrait être:
" la probabilité que la cause d'un incident soit diagnostiquée ou que la faute soit corrigée étant donné un effort prédéfini et un environnement donné ".
Comme pour l'utilité, ceci est trop difficile à mesurer.

3.4.1 La mesure des attributs externes de la maintenabilité.

La maintenance est un attribut externe car elle dépend, en plus du produit, de la personne qui maintient et du processus établit de maintenance. Si on considère la vue externe de la maintenance, la mesure la plus commune est le temps moyen pour faire un changement.

On a besoin pour cela de connaître:

- le temps pour reconnaître un problème.
- le temps passé à des activités de management concernant le problème.
- le temps pour collecter des outils de maintenance.
- le temps pour analyser le problème.
- le temps pour spécifier le changement.
- le temps pour effectuer le changement (incluant les tests et reviews[13]).

D'autres mesures utiles sont:

- le temps moyen pour corriger un bug.
- le nombre de problèmes non résolus.
- le pourcentage des changements qui introduisent de nouvelles fautes (qui est en général à un peu plus de 10%).
- le nombre de modules modifiés pour faire un changement.
- le nombre de problèmes reportés.
- la densité des fautes après que le produit ait été livré.
- le pourcentage de code modifié (qui donne une idée de la stabilité du code).

3.4.2 La mesure des attributs internes qui influencent la maintenabilité.

Les attributs internes influencent la maintenance. C'est le cas des métriques de complexité[14]. Pour Mac Cabe, aucun module ne devrait avoir une complexité cyclomatique plus grande que 10.

La mesure Vinap[15] apparaît plus sensible pour identifier les problèmes de maintenance. Pour Bache, un produit avec une valeur de Vinap de plus de 100 fera de la maintenance un vrai cauchemar (d'après ses propres termes) [BACH88].

Nous venons de voir que la maintenance dépend en partie de la complexité du logiciel. De plus, il est évident qu'un produit peu structuré et peu documenté sera difficile à maintenir.

De même, la lisibilité est particulièrement importante.
La lisibilité d'un document peut être mesurée à partir du "Fog Index", définit par Gunning:

F= 0.4*[(nbre de mots/nbre de phrases) + (pourcentage de mots de 3 syllabes ou plus)].

La lisibilité décroît quand F augmente.

---

[13] ce terme est défini en 6.3.
[14] voir en 2.2.3.
[15] voir en 2.2.3

Pour mesurer la lisibilité du code source, la mesure de Young peut être utilisée:

R = 0,295*a - 0,499*b + 0,13*c où

R est la lisibilité,
a est la moyenne normalisée de la longueur des variables,
b est le nombre de lignes contenant des instructions,
c est la complexité cyclomatique de Mac Cabe.
Les paramètres ont été évalués subjectivement à partir d'une analyse de régression.

Même si les attributs internes peuvent être des indicateurs importants, il ne faut pas surestimer leur influence sur la maintenabilité.
Pourtant, Belady et Lehman ont proposé le modèle suivant où la complexité est un paramètre majeur:

$M = p + K e^{c-d}$ où

M est une mesure de l'effort total dépensé en maintenance,
p est l'effort productif,
K est une constante empirique,
c est une mesure de la complexité
et d est une mesure du degré de familiarité avec le logiciel.

Un tel modèle donnera certainement des mesures grossières.

CHAPITRE 4:
LES MODÈLES DE COÛTS ET DE LA PRODUCTIVITÉ.

Vu les coûts exorbitants qu'engendrent actuellement le développement des logiciels, il est important de prévoir le coût et la productivité du développement d'un logiciel. Ces deux notions sont fortement liées. En effet, on cherche à augmenter la productivité, en partie pour réduire les coûts. De plus, les ressources humaines sont une notion primordiale aussi bien pour l'estimation des coûts (c'est le personnel qui coûte le plus cher dans le développement d'un logiciel) que de la productivité (on cherche en effet à estimer la productivité du personnel).
En 4.1 sera présenté le problème de l'estimation des coûts et en 4.2, un aperçu de ce qu'on entend par productivité sera donné.

4.1 Coûts et Estimation des Coûts.

Par estimation des coûts, on entend en fait l'estimation de l'effort et du temps pour développer un projet (on pourra ainsi évaluer combien le projet va coûter car les dépenses principales sont relatives au personnel). Afin de donner une indication de la précision, l'estimation des coûts doit être basée sur 3 nombres: la valeur la plus vraisemblable et les bornes inférieures et supérieures.

En 4.1.1, nous introduirons au problème de l'estimation des coûts en montrant son importance et sa difficulté. 4 méthodes d'estimation seront présentées. La partie 4.1.2 sera consacrée au modèle COCOMO qui identifie 15 attributs importants en plus de la taille pour mesurer l'effort. En 4.1.3, un aperçu du modèle SLIM de Putnam sera donné. Pour conclure, en 4.1.4 les problèmes des méthodes existantes et les solutions possibles seront présentées.

4.1.1 Vue générale de l'estimation des coûts.

Les coûts sont estimés tout au long du développement d'un logiciel. En faisant une analyse coût/bénéfice, il est possible de déterminer si un projet est faisable (ces estimations préliminaires sont les plus difficiles à déterminer car il y a peu d'informations connues sur le projet). Une fois que le projet est commencé, une estimation plus détaillée est nécessaire pour effectuer le planning (la durée et l'effort réel des tâches effectuées sont comparés avec les valeurs prévues).

Dans un projet, une limite financière à ne pas dépasser est souvent fixée. Il ne faut pas confondre cela avec l'estimation des coûts, même si l'estimation peut servir à fixer la limite. En fait, l'estimation devrait être utilisée pour voir si, à un état du projet, la limite n'est pas dépassée.

L'estimation des coûts d'un logiciel est plus difficile que pour le cas des autres industries où on fabrique toujours le même type de produit. Ceci est dû au fait que dans l'industrie des logiciels, de nouveaux produits sont habituellement conçus. Des estimations seront possibles mais elles ne seront pas très précises. Comme l'industrie du logiciel est en général assez mauvaise pour collecter des données sur les projets passés, il est difficile d'approcher la taille du futur produit et sa productivité (ce qui constituent des paramètres importants pour estimer les coûts).

Il y a 4 méthodes pour estimer les coûts:

1. L'opinion d'expert: cela consiste à se baser sur l'expérience personnelle.

2. L'analogie: c'est une approche plus formelle que l'opinion d'expert. L'effort d'un projet passé similaire est pris comme estimation initiale du nouveau projet et s'en suit une ajustation en fonction des différences entre le projet passé et le nouveau.

3. La décomposition: cela consiste à décomposer un projet en des tâches de plus bas niveau. On estime alors l'effort requis pour exécuter ces tâches et on somme pour avoir une estimation pour le projet.

4. L'utilisation d'équations: cela consiste à calculer l'effort à partir d'équations mathématiques.

Toutes ces méthodes peuvent être utilisées dans une approche top-down et bottom-up sauf la décomposition qui est spécifique uniquement à une approche bottom-up.
Dans une approche bottom-up, on estime l'effort des tâches individuelles et on somme pour avoir une estimation de l'effort du projet tout entier.
Dans une approche top-down, on fait une estimation du projet tout entier puis on déduit une estimation de l'effort des tâches particulières en assumant que c'est un pourcentage de l'effort total.

Remarque: Ni « price to win » ni « parkinson's law » [DEMA87] ne sont des méthodes d'estimation. Ce sont plutôt des cibles à atteindre. "Price to win" consiste à imposer des contraintes au projet. "Parkinson's law" consiste à établir un calendrier irréaliste, s'imaginant à tort que les programmeurs aiment les challenges et qu'ils ne travaillent pas durs s'ils ne sont pas soumis à des pressions intenses.

4.1.2 Un modèle composite pour estimer l'effort : COCOMO.

Un modèle composite incorpore une combinaison d'équations analytiques, de données trouvées statistiquement et des jugements d'experts. Le modèle composite le plus connu est COCOMO (Constructive Cost Model), développé par Boehm en 1981 [BOEH81],[BOEH84]. C'est le plus complet et le plus documenté de tous les modèles pour estimer l'effort. Il fournit des formules spécifiques pour estimer le temps de développement, l'effort de tout le développement et l'effort par phase et activité.

Les équations de COCOMO sont dérivées d'un base de données constituée à partir de 63 projets, établis en 15 ans à TRW Système, Inc. Les projets ont une taille variant de 2.000 à 1.000.000 lignes de codes. Ils ont été écrit en plusieurs langages, sont de type différents (scientifique, gestion, etc). Les équations de COCOMO ne sont pas obtenues directement par une méthode de régression mais en utilisant une combinaison d'expériences, de résultats d'autres modèles d'estimation du coût et d'opinions subjectives de managers.
Dans ce qui suit, un aperçu est donné des équations et attributs pour estimer l'effort. S'en suivra une critique générale de COCOMO.

4.1.2.1 Les équations pour estimer l'effort durant le développement.

COCOMO définit 3 niveaux de détail (basic, intermédiaire et détaillé) ainsi que 3 modes de développement qui sont les suivants:

- Les projets de mode organique sont relativement petits en taille, demandent peu d'innovation, ont des contraintes de livraison relativement laxistes et un environnement de développement stable.

- Les projets de mode fixé (embedded) sont de taille relativement large, avec des contraintes strictes, un degré élevé de hardware, une interface complexe avec le client, des spécifications strictes et un besoin plus grand d'innovation.

- Les projets du mode semi-détaché tombe entre le mode organique et fixé.

Les équations pour estimer l'effort sont de la forme:

$E = Ai(S\ e^{Bi}) * M(x)$ où

E représente l'effort en personne-mois,
S est la taille mesurée en KLoc,
M(x) est un multiplicateur composite qui dépend de 15 attributs orientés coûts (cost drivers), qui seront présentés par la suite.
Ai est fonction du mode et du niveau que l'on vient de présenter ci-dessus et Bi est fonction du mode seulement. Le *tableau 4.1* donne les valeurs de Ai et Bi pour les niveaux basic et intermédiaire et les 3 modes.

Niveau	Basic		Intermédiaire	
Mode	**Ai**	**Bi**	**Ai**	**Bi**
Organique	2,4	1,05	3,2	1,05
Semi-détaché	3,0	1,12	3,0	1,12
Fixe	3,6	1,20	2,8	1,20

Tableau 4.1: Les valeurs des paramètres Ai et Bi.

4.1.2.2 Les 15 attributs orientés coûts.

COCOMO utilisent 15 attributs orientés coût. Ils sont dérivés de 29 attributs développés par Walston et Félix. Ils sont répartis en 4 catégories.(voir *Figure 4.1*).

Attributs Produit

RELY: Reliability Requise du Logiciel.
DATA: Taille de la Base de Donnée.
CPLX: Complexité du Produit.

Attributs de l'ordinateur

TIME: Contrainte au niveau du Temps d'Exécution.
STOR: Contrainte au niveau de la Mémoire Centrale.
VIRT :Volatilité de la Machine Virtuelle.
TURN: Temps de Turnaround de l'Ordinateur.

Attributs du Personnel

ACAP: Capacité de l'Analyste.
AEXP: Expérience de l'Application.
PCAP: Capacité du Programmeur.
VEXP: Expérience de la Machine Virtuelle.
LEXP: Expérience du Langage de Programmation.

Attributs du Projet

MODP: Pratiques de Programmation Modernes.
TOOL: Utilisation des Outils de Logiciel.
SCED: Schéma de développement Requis.

Figure 4.1: les attributs orientés coût.

Les facteurs omis par Boehm sont le type de l'application, le niveau du langage, la qualité du management, la complexité de l'interface avec le client, les restrictions au niveau de la sécurité.
Boehm utilise une approche « heuristique » pour déterminer l'effet de chaque attribut sur l'effort.

La procédure pour déterminer M(x) est la suivante:

- On assigne d'abord un taux à chacun des attributs décrits ci dessus (*Figure 4.1*), grâce à un ensemble de tables. Par exemple, pour déterminer AEXP et STOR on dispose du *tableau 4.2*:

Attributs orientés coûts

Taux	AEXP (expérience)	STOR (% de mémoire)
Très bas	<= 4 mois	-
Bas	1 an	-
Nominal	3 ans	<= 50
Haut	6 ans	70
Très haut	12 haut	85
Extrèmement haut	-	95

Tableau 4.2: Taux de AEXP et STOR.

- Ensuite, grâce à d'autres tables, on obtient la valeur numérique du taux. Le *tableau 4.3* reprend les valeurs numériques des taux de certains des 15 attributs orientés coût. Les taux nominaux ont tous une valeur numérique égale à 1.

	Taux						
Attributs	Très bas	Bas	Nominal	Haut	Très haut	Extrèmement haut	**Domaine de productivité**
AEXP	1,29	1,13	1,00	0,91	0,82	-	**1,57**
STOR	-	-	1,00	1,06	1,21	1,56	**1,56**
CLPX	0,70	0,85	1,00	1,15	1,30	1,65	**2,36**
ACAP	1,46	1,19	1,00	0,86	0,71	-	**2,06**
PCAP	1,42	1,17	1,00	0,86	0,70	-	**2,03**

Tableau 4.3: Valeurs numériques des taux.

M(x) est obtenu en multipliant la valeur obtenue des taux des 15 attributs.

Chaque attribut a un domaine de productivité. Il est obtenu en divisant le plus grand taux par le plus petit.

Ainsi, pour AEXP (expérience de l'application), le domaine de productivité est de: 1,29/0,82 = 1,57. Un produit avec un taux de AEXP très faible aura une productivité de 157 % inférieure à un produit avec un taux très haut.
Les attributs avec les plus grands taux sont la complexité du produit puis les capacités de l'analyste et du programmeur. Certains de ces attributs peuvent être contrôlés par le management (la capacité du programmeur par exemple) alors que d'autres ne peuvent pas être contrôlés comme la reliability requise ou la complexité du produit.
Remarquons que ces 15 attributs orientés coûts ont aussi un impact important sur la productivité, qui sera abordée en 4.2.

4.1.2.3 Les équations pour estimer le temps.

Les équations pour déterminer le temps de développement (en mois) sont les suivantes:

équation 4.1: $T = 2{,}5\ e^{0{,}38}$ pour un mode organique.
équation 4.2: $T = 2{,}5\ e^{0{,}35}$ pour un mode semi-détaché.
équation 4.3: $T = 2{,}5\ e^{0{,}32}$ pour un mode fixé.

Ces équations sont d'une forme similaire à celles obtenues par Walston et Felix [WALS77] et Putnam [PUTN78].

4.1.2.4 Critique de COCOMO.

Le problème le plus important de COCOMO est le nombre important de paramètres, ce qui donne une trop grande flexibilité et un manque de précision. Par exemple, pour calculer M(x), différents analystes pourront déterminer M(x) de telle manière que l'on obtienne des estimations variants de 0,088 à 7,2. Ceci est dû à la subjectivité de l'analyste. Les 15 paramètres peuvent alors constituer une faiblesse. On devrait alors essayer de réduire le nombre de paramètres, en combinant plusieurs attributs en un seul. D'ailleurs, les attributs sont hautement corrélés en pratique, même s'ils ne le sont pas en théorie.

Une deuxième critique est le fait que les constantes du modèle ont été obtenues empiriquement, à partir d'une base de données spécifique. On ne peut être sûr d'obtenir de bonnes prévisions à partir de ces données pour un autre projet.

4.1.3 Un modèle de contraintes: Slim.

En 1978, Putnam a développé SLIM, un modèle de contraintes. Un modèle de contraintes met en relation 2 ou 3 paramètres de coût comme l'effort, la durée ou le nombre de personnes travaillant sur le projet. Slim permet d'estimer l'effort total et le temps pour les grand projets (plus de 70000 Loc).
Le modèle de Putnam est basé sur les courbe de Rayleigh (voir *figure 4.2*) qui décrivent l'évolution du nombre de personnes travaillant sur un projet au cours du temps.

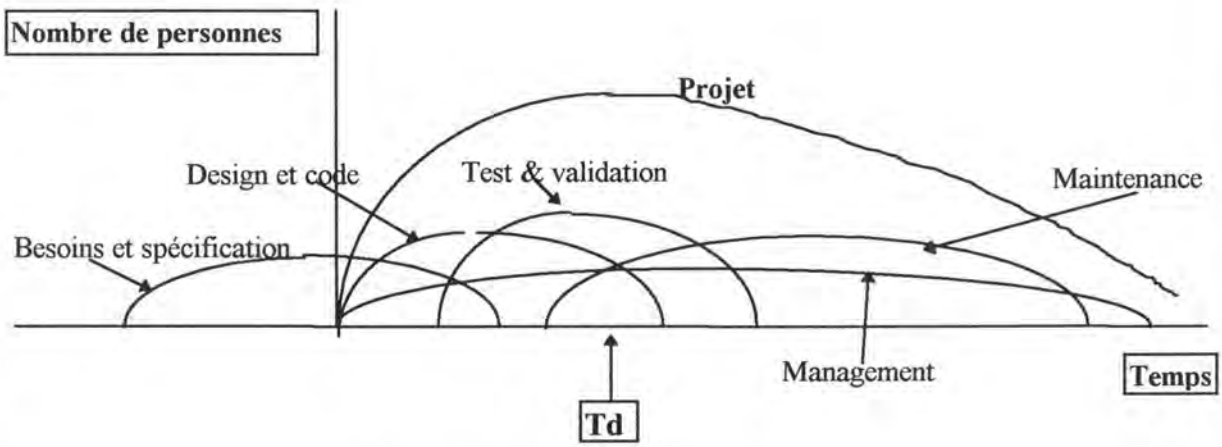


Figure 4.2: les courbes de Rayleigh.

Comme le montre la *figure 4.2*, il y a en fait 4 sous-courbes correspondant aux phases suivantes: design et code, test et validation, maintenance, management. La courbe de Rayleigh du projet est la somme de ces 4 sous-courbes.

Putnam se base sur l'équation de base de la courbe de Rayleigh et dérive une relation appelée la « software équation », qui mesure la taille en Loc en fonction d'un facteur technologie C, de l'effort total du projet K (qui inclut l'effort de maintenance) et le temps écoulé depuis le début du projet jusqu'à la livraison du produit (c'est à dire le temps de développement).

équation 4.4: $Taille(en\ Loc) = C * (K\ e^{1/3}) * (Td\ e^{4/3})$

avec K en personnes/année, Td en année.

Sur la courbe de Rayleigh, Td est le point où la courbe atteint un maximum. C, le facteur technologie peut prendre 20 valeurs différentes.

Putnam introduit une autre équation pour estimer l'effort ou la durée:

équation 4.5: $D_0 = \frac{K}{Td\ e^3}$ où

Do est une constante appelée "manpower acceleration". Elle prend une valeur spécifique pour un type particulier de logiciel. Par exemple, $D_0 = 7,3$ pour un nouveau logiciel avec beaucoup d'interfaces et d'interactions avec d'autres systèmes; $D_0 = 15$ pour les systèmes fermés; $D_0 = 27$ pour la réimplementation de systèmes existants. Ces valeurs sont obtenues à partir de systèmes existants.

L'équation 4.6 est obtenue en combinant les 2 équations 4.4 et 4.5:

équation 4.6: $K = [(S \div C)\, e^{9/7}] * [(D_0)\, e^{4/7}]$.

4.1.3.1 Les points de similarité entre COCOMO et SLIM.

On remarque des points de similarité entre le modèle Slim de Putnam et le modèle Cocomo de Boehm. L'équation 4.5 du modèle Slim montre que le temps est proportionnel à la racine cubique de l'effort, tout comme les équations 4.1, 4.2, 4.3 de Cocomo. L'équation 4.6 de Slim montre que l'effort est proportionnel à la taille à la puissance 9/7 (c'est à dire 1,286), ce qui est une valeur proche de celles du modèle Cocomo (voir les valeurs de Bi dans le *tableau 4.1*).

4.1.3.2 Critique de SLIM.

Tout comme Cocomo, le modèle Slim a été critiqué. Tout d'abord, il faut remarquer que la spécification des besoins n'est pas repris dans ce modèle (sur la *figure 4.2*, la courbe relative au projet n'inclut pas celle relative aux spécifications des besoins). De plus, les courbes de Rayleigh ne sont pas communément acceptées. Parr [PARR81] suggère que le design et le code ne commence pas avec 0 personne et propose un ajustement à la courbe. Dans Slim, le facteur technologie est difficile à estimer précisément mais il se trouve que ce modèle est très sensible à ce facteur.

4.1.4 Les problèmes avec les méthodes existantes.

En général, les prévisions des modèles proposés sont plutôt mauvaises. En effet, en évaluant les modèles selon 2 approches, on a constaté:

1. En appliquant le modèle à des données d'un projet passé, dans la plupart des cas, l'effort prédit est très différent de l'effort réel.

2. Si on utilise une série de modèles pour estimer l'effort pour un projet particulier, on s'aperçoit que les modèles donnent des résultats très différents.

Comme il l'a déjà été dit, les facteurs trop nombreux, sensés permettre de représenter au mieux les spécificité de l'organisation, deviennent un handicape quand leur estimation subjective ne peut être faite précisément.

Beaucoup de modèles utilisent la taille comme paramètre mais il est très difficile de la mesurer au début du développement d'un logiciel, à partir d'une spécification des besoins par exemple.

En fait, la mauvaise performance des modèles est due principalement au fait qu'ils sont développés à partir de l'analyse d'un ensemble de données et qu'ils incorporent les caractéristiques particulières de ces données.
Un moyen de remédier à ce problème est de développer un modèle local d'estimation des coûts, c'est à dire un modèle propre à son organisation et à son projet. Différentes méthodes d'estimation des coûts pourront être utilisées mais on formulera ses propres équations à partir des données collectées sur les projets passés de l'organisation. Il sera utile d'avoir des feedbacks, puisqu'ils permettent d'améliorer la technique d'estimation quand de nouveaux événements surviennent.

4.2 Productivité et Estimation de Productivité.

Lors de la révolution industrielle, la productivité a été augmentée en divisant le travail, en standardisant, etc.
De même, pour les logiciels, les managers essaient depuis une dizaine d'années d'améliorer la productivité pour d'une part combler le trou entre l'offre et la demande en programmeurs et d'autre part pour réduire les coûts de développement des logiciels qui deviennent alarmants.

En 4.2.1, l'équation générale pour mesurer la productivité sera présentée. Les facteurs qui influencent la productivité seront identifiés en 4.2.2. La section 4.2.3 présentera les conclusions de l'étude menée par Walston et Felix. Nous verrons ensuite en 4.2.4 comment la programmation structurée peut influencer la productivité et enfin en 4.2.5 comment la taille de l'équipe agit sur la productivité. Une série d'équations sera présentée: ces équations permettent d'évaluer l'interaction au sein de l'équipe et d'approximer sa taille idéale.

4.2.1 L'équation générale de la productivité.

Pour mesurer la productivité, l'équation suivante est généralement utilisée:

Productivité = Taille en Loc / Effort en programmeurs * mois .

Remarque: en général, la productivité varie entre 20 et 1500 Loc/personne-mois.

Une première limitation de cette définition est qu'elle ne peut être appliquée que pour la phase de code. En effet, cette équation ne pourra être utilisée pour la phase de test ou de design puisqu'on ne génère pas de code durant cette phase.

Mais plus important encore, cette définition ne capture pas notre compréhension intuitive de la productivité. Dans l'industrie automobile, compter le nombre de voitures produites par heures fournit une bonne mesure de la productivité car la duplication est une tâche majeure. Ce n'est pas le cas pour la production de logiciel (la duplication n'est pas ce qui coûte le plus cher!) où on cherche à augmenter la productivité des ressources, principalement des programmeurs (remarquons quela productivité d'un

outil comme un compilateur peut aussi être mesurée. Par exemple, pour faire un choix entre 2 compilateurs, on pourra les comparer selon leur temps de compilation, leur prix, etc).

Une définition de la productivité devrait tenir compte de la qualité du code généré et pas uniquement de sa taille (tout comme on ne peut mesurer la productivité d'un poète seulement en comptant le nombre de lignes qu'il a écrit).

Néanmoins on ne doit pas conclure pour autant qu'il ne faut pas utiliser cette équation qui peut avoir son utilité mais qu'il faut manier avec précaution.

Il faut d'abord savoir ce qu'on définit comme Loc et comment l'effort est calculé (voir chapitre 2). Tient-t-on seulement compte du temps pour coder ou alors inclus-t-on la maintenance, l'analyse des besoins, etc dans l'effort?
Pour un même produit, la productivité variera selon les définitions choisies. Et même en supposant que l'effort et les Loc peuvent être définis précisément, un problème sérieux est qu'on ne peut comparer la productivité de produits écrits dans des langages différents: les taux de productivité de langages de haut niveau sont toujours plus faibles que ceux en langage assembleur (mais il y a plus de Loc à produire en assembleur qu'en pascal par exemple, pour un même projet).

Pour résoudre ce problème de comparabilité, on peut convertir un nombre de Loc en langage de haut niveau en un nombre équivalent de lignes d'objets grâce à un ratio qui dépend du langage. Mais ses ratios sont des moyennes et cette approche est limitée car elle ne tient pas compte du type de programme. Une étude à Boeing Computer Service a montré que la productivité moyenne dépend du type de logiciel :

Type de logiciel	**Productivité en Loc/Personnes-Mois**
Mathématique	167
Commercial	125
Logique	83
Temps réel	25

Tableau 4.4: Productivité en fonction du type de logiciel.

D'autres métriques peuvent être utilisées pour mesurer la taille du code comme celles définies par Halstead ou les functions points - FP- de Albrecht (voir le chapitre 2 sur les métriques).
Les FP peuvent être utilisées à toutes les étapes du cycle de vie d'un logiciel mais elles sont difficiles à calculer. C'est pourquoi les managers préfèrent utiliser les Loc qui sont plus beaucoup plus facile à appréhender.

Même si cette équation a ses limites, elle est toujours beaucoup utilisée car:
- elle est facile à calculer (ce qui ne justifie pas son utilité).
- elle permet des comparaisons (limitées, comme on l'a fait remarquer).

Une meilleure définition de la productivité est attendue.

4.2.2 Les facteurs affectant la productivité.

Pour améliorer la productivité, il faut pouvoir identifier les facteurs qui l'affectent mais aussi avoir des mesures raisonnables de ces facteurs. L'étude SDC de 169 projets [NELS66] identifie 104 facteurs. Voici certains de ces facteurs, classés par catégorie:

- Facteurs personnel: capacités individuelles, années d'expérience, expérience du langage de programmation, expérience avec des cas similaires, la taille de l'équipe, l'organisation de l'équipe, l'expérience de l'équipe travaillant ensemble, la qualité du management, etc.

- Facteurs Processus: le langage de programmation, l'utilisation de programmation structurée, inspection du code, outils de test, compilateurs optimisés, système de B.D., etc.

- Facteurs Produit: taille du produit, reliability, besoin en temps réel, structure de données, nombre de modules, couplage entre les modules, type de logiciel, quantité de code réutilisé, etc.

- Facteurs ordinateurs: temps de réponse, contrainte en temps, en mémoire, etc.

Certains facteurs sont dépendants d'autres. Comme il est impossible d'inclure tous ses facteurs dans un modèle de productivité, il faut réduire la liste des facteurs et ne garder que ceux qui ont les propriétés suivantes:

1. mesurabilité et objectivité: il faut obtenir des taux de productivité indépendants de la personne qui effectue la mesure.

2. généralité: par exemple, les restrictions de sécurité ne sont pas applicables à tous les types de logiciels. Ce facteur est sans intérêt pour un modèle général de productivité.

3. significatif: en utilisant des méthodes statistiques, si on voit qu'un facteur agit peu sur la variance, on le rejette.

4. indépendance: les facteurs corrélés doivent être groupés et représentés par un seul facteur.

Des études ont montré que l'effet de la capacité individuelle est très grand pour les petits projets avec un programmeur (dans un ordre de 1 à 26). Pour les grands projets, cet effet est moins prononcé (dans un ordre de 1 à 4).
L'expérience de l'équipe travaillant ensemble à un effet significatif. Les programmeurs expérimentés ont une productivité de 2 à 4 fois plus grande que les non-expérimentés.

L'étude de Walston-Felix est intéressante puisqu'elle a permis d'identifier des facteurs importants agissant sur la productivité.

4.2.3 L'étude de Walston-Felix sur la productivité.

Au début des années 1970, Walston-Felix, à partir de 60 projets, ont identifié 29 facteurs corrélés de manière significative avec la productivité (une régression multi-linéaire a réduit 68 facteurs à 29, ceux qui ne sont pas repris ne faisant pas varier beaucoup la productivité).
La productivité est mesurée en lignes de codes source délivrés/personnes*mois.
Pour chaque facteur, un ordre de productivité est calculé en effectuant le rapport entre la productivité maximale et la productivité minimale.
Ceci met en évidence les facteurs qui ont le plus grand impact sur la productivité. Ce sont les suivants:

Facteurs	**Ordre de Productivité**
Complexité de l'interface avec le client	4,03
Changements lors du design	2,94
Expérience du client avec le domaine d'application du projet	2,84
Expérience du personnel et qualification	3,11
Expérience du langage de programmation	3,16
Expérience avec une application de taille et complexité similaire ou plus grande	2,81
Nombre de pages délivrées pour 100 LOC	2,56

Tableau 4.5: Ordre de productivité de facteurs.

Ainsi la productivité d'un projet A pourra être 3 fois plus grande que celle d'un projet B, A et B étant identiques excepté pour le nombre de changements lors du design.

On voit à partir de cette étude que les facteurs qui influencent le plus la productivité sont ceux en rapport avec le personnel.
Des facteurs comme la complexité du flux du programme (ordre de 1,43), l'inspection du design et du code (ordre de 1,54) ont peu d'impact.

Une étude menée par ITT (International Telephone and Telegraph Coorporation) a abouti aux mêmes conclusions que l'étude de Walston-Felix. Les seuls points de divergence sont la participation du client pour la spécification des besoins et l'expérience du client avec l'application. Cette étude a réduit 13 facteurs qui agissent sur la productivité et a mis en évidence que la productivité augmentera seulement si le management implémente un programme intègre avec des techniques, pratiques et outils qui supportent le processus de développement en totalité. Un facteur pas assez contrôlé fera baisser la productivité alors qu'un seul facteur, quel que soit la manière dont il est contrôlé ne pourra permettre un gain de productivité à lui seul.

4.2.4 La programmation structurée et ses effets sur la productivité.

Un design et une programmation structurée permettent de réduire les erreurs, d'augmenter la clarté du programme et de fournir moins d'effort pour la maintenance. Mais qu'en est-il pour la productivité?

En 1980, Brooks [BROO80], à partir de la B.D. fournies par Walston-Felix, a conduit la première étude qui quantifiait le gain potentiel de la programmation structurée pour la productivité.
En classant les projets selon leur structuration, Brooks a conclut:

1- Pour les projets non structurés, on a souvent une faible productivité à cause d'une taille élevée, une interface complexe avec le client, une complexité importante de l'application, des contraintes en temps ou en mémoire ou encore une faible expérience du personnel.

2- Mais pour les projets structurés, aucune de ces contraintes n'affecte la productivité, excepté la taille du projet (on remarque que la productivité décroît exponentiellement quand la taille du projet augmente). La structuration permet de surmonter les effets négatifs des facteurs cités auparavant.

3- La programmation structurée permet un gain de productivité (plus important pour les projets de grande taille).

4.2.5 Les effets de la taille sur la productivité.

Plusieurs études révèlent que la productivité individuelle baisse quand la taille de l'équipe d'implémentation augmente. Brooks dans "The Mythical Man-Month" [BROO75] cite 2 raisons pour expliquer ce phénomène:

1. Quand la taille de l'équipe augmente, il y a un besoin de coordination plus important, ce qui nuit à la productivité.

2. Les membres ajoutés à une équipe doivent maîtriser le design du projet avant de pouvoir y contribuer. C'est pourquoi Brooks affirme: " Si on met du personnel supplémentaire sur un projet en retard, il sera encore plus en retard ". Même si cette citation est communément acceptée, Fenton [FENT91] affirme que la productivité ne baissera pas nécessairement mais le code produit sera de moins bonne qualité. En effet, si les programmeurs ont l'impression qu'on mesure leur productivité, ils pourront facilement l'augmenter (avec des commentaires, des lignes blanches, etc).

Il est surprenant que ni Walston-Felix, ni Boehm dans son modèle COCOMO, n'incluent la taille du projet comme facteur.
Remarquons qu'à propos de COCOMO, les 15 facteurs identifiés comme ayant un impact sur le coût (voir en 4.1) ont aussi un impact sur la productivité puisque ces 2 notions sont corrélées.

A partir d'une B.D. de 187 projets, on a pu mettre en évidence que la productivité décroît exponentiellement avec la taille de l'équipe.

Ceci est mis en évidence par la formule suivante:

$\hat{L} = 777\ \overline{P}\, e^{-0,5}$ avec un erreur standard de 0,07 (dans un plan log-log).

P est le niveau moyen en personnel.

$\overline{P} = E$ (format en personne * mois) ÷ T (durée du projet en mois).

$\hat{L}$ est la productivité estimée.

Ce modèle est précis puisque 80% des valeurs qu'il prédit correspondent à la valeur exacte plus ou moins l'erreur standard.

Un modèle plus théorique peut être élaboré, à partir de l'observation de Brooks ([BROOK75]):
"Quand la taille de l'équipe augmente, le nombre de chemins de communication humaine tend aussi à augmenter pour permettre la coordination entre les modules du programme".

Si chaque membre d'une équipe de taille P peut coordonner ses activités avec chacun des autres membres, il y a P(P-1)/2 chemins (Il y a P membres et chaque membre communique avec P-1 membres. Comme un chemin permet une communication dans les 2 sens, on doit diviser P*(P-1) par 2).

Par contre, si chaque membre ne peut communiquer qu'avec un seul supérieur, on aura P-1 chemins.

Entre ces 2 extrêmes, (un cas avec une communication maximale et une complexité $O(P\,e^2)$, un cas avec une communication minimale et une complexité $O(P)$), le management peut sélectionner une organisation avec une complexité $O(P^{\beta})$ avec $1 \leq \beta \leq 2$.
Par exemple, si l'équipe est partitionnée en K groupes de même taille avec une communication complète dans chaque groupe mais un seul chemin de communication par groupe avec un supérieur, $C(P)$, le nombre total de chemins de communication, sera:

$$C(P) = K * \frac{[(P \div K) * ((P \div K) - 1)]}{2} + K$$

En résolvant l'équation $P^{\beta} = C(P)$, on trouve β.
β proche de 1 indique une interaction minimale.
β proche de 2 indique une interaction maximale.
En pratique, un manager essaiera d'avoir une valeur de β autour de 1,5.

En supposant qu'en moyenne, chaque membre d'une équipe peut produire L Loc par mois et que chaque chemin résulte en une perte nette de l Loc par mois, chaque programmeur produira en moyenne:

$\overline{L} = L - l(P-1)e^{\delta}$ avec $0 < \delta \leq 1$.

δ donne une idée du nombre de chemins disponibles pour un programmeur. δ = 1 indique que chaque membre peut communiquer avec chacun des autres membres.

La productivité du groupe sera:

$$L_{group} = P * \overline{L} = P[L - l(P-1)e^{\delta}$$

Alors que P augmente, L commence par augmenter puis arrive à un pic et décroît.
Afin de trouver la valeur de P correspondant à la valeur de L_{group} optimale, l'équation ci-dessus est dérivée:

$$\frac{dL_{group}}{dP} = [L - l(P-1)e^{\delta}] - [l * P * (P-1)e^{\delta-1}] = 0 \Leftrightarrow (P-1)e^{\delta} * [1 + (\delta * P) \div (P-1)] = L \div l$$

Connaissant L, l et δ, on peut trouver P optimal. L'équation $L_{group} = P\overline{L}$ permet de connaître la productivité optimale du groupe.

Ces équations peuvent être considérées comme un modèle de productivité. L, l, δ peuvent être déterminés à partir d'une B.D. Néanmoins ce modèle est limité car il n'inclut pas tous les facteurs qui influencent la productivité. Son utilité est de sensibiliser les managers sur la composition des groupes.

CHAPITRE 5:

DATRIX

(UN OUTIL PRODUISANT DES MÉTRIQUES).

5.1. Introduction.

Dans ce chapitre, Datrix, un outil destiné à mesurer la qualité d'un logiciel en générant des métriques, est présenté.
Datrix a été développé à l'école polytechnique de Montréal et a pour buts principaux:

- De vérifier la cohérence et la consistance d'un projet dans son ensemble.
- D'évaluer la qualité du code source du projet à travers les métriques.
- D'alerter l'utilisateur de la valeur "anormale" d'une métrique. Pour cela, il faut définir un fichier "profile" où à chaque métrique est assignée une borne inférieure et une borne supérieure. La valeur d'une métrique est considérée "anormale" si elle n'est pas comprise entre les 2 bornes correspondantes.

Datrix est un outil destiné à être utilisé par:

- Les personnes impliquées dans la mesure de qualité de logiciel (SQA).
- Les ingénieurs de logiciel.
- Les analystes et les programmeurs responsables de la documentation des logiciels et des tests.
- Les personnes qui s'occupent de la maintenance de logiciels.

Datrix génère 2 graphes:

- Le graphe d'appel de Datrix qui représente l'organisation des unités (une unité est une partie de programme comme une fonction, une routine, une procédure, la partie principale du programme). Ce graphe fournit la même information qu'un callgraphe[16] et permet de modéliser la structure du système. Le graphe d'appel met en évidence le nombre de fois qu'une unité appelle ou est appelée par une autre.
- Le graphe de contrôle.

Ce concept fondamental de Datrix est présenté en 5.2. En 5.3, les métriques générées par Datrix sont expliquées. Une critique s'en suit en 5.4.

5.2. Le Graphe de Contrôle.

Après avoir défini ce qu'est un graphe de contrôle (5.2.1), il faudra voir comment on l'élabore à partir du code source (5.2.2).

5.2.1 Définition.

Pour chaque unité de code, un graphe de contrôle (voir *figure 5.3*) est généré par Datrix. Il permet de modéliser la structure du code, de mettre en évidence les chemins possibles et les instructions de contrôle (comme les boucles, les conditionnelles).
Même si le graphe de contrôle est constitué de sommets et d'arcs, sa représentation est très différentes de celle commune à la théorie des graphes.

Un sommet correspond à une ligne du fichier source et est représenté par une ligne horizontale précédée par le numéro de ligne. C'est un endroit où les arcs sont connectés.
Certains sommets sont spéciaux et ont une signification particulière:

Les sommets d'entrée où commence l'exécution de l'unité de programme. Ils sont marqués ainsi: ->.

- Les sommets de sortie où se termine l'exécution de l'unité de programme. Ils sont marqués ainsi: <-.

[16] voir 2.4.1.

Un arc est un lien direct entre 2 sommets. Un arc montre la façon d'aller d'un sommet à un autre tout en exécutant un groupe d'instructions. Il y a 2 types d'arc:

- Les arcs en avant: ils sont représentés par une ligne verticale simple, indiquant que l'on doit suivre l'arc vers le bas, comme on lit normalement un fichier source.
- Les arcs en arrière: ils sont représentés par une double ligne verticale, indiquant que l'on doit remonter le long de l'arc, comme pour le retour d'une boucle.

Quand un arc croise une ligne horizontale, il y a un croisement. 2 types de croisements sont à considérer:

- Ceux qui respectent les principes de la programmation structurée (voir la remarque ci-dessous). Ils sont simplement appelés croisements et sont représentés par une ligne de croisement.
- Ceux qui ne respectent pas les principes de la programmation structurée. Ils sont appelés branches de structure et sont représentés par un carré noir au point de croisement.

Remarques: Avec Datrix, la règle générale de programmation est que chaque structure de contrôle[17] doit avoir une et une seule entrée et une et une seule sortie .

5.2.2 L'élaboration du graphe de contrôle.

La correspondance entre les instructions de contrôle du langage de programmation et le graphe de contrôle est fondamentale dans la construction du graphe de contrôle.
Il faut remarquer que les métriques relatives au graphe dépendent de la façon dont le graphe est dérivé des instructions du code source. En conséquence, on a établit des règles formelles pour accomplir les transformations automatiquement.
Avant de présenter le processus d'élaboration du graphe de contrôle, il faut définir une notion qui sera utilisée: les éléments primaires.

5.2.2.1 Les éléments primaires.

Le code des programmes écrits en langage de programmation procéduraux peut être décomposé en segments de base appelés éléments primaires. Ces segments deviendront des arcs du graphe de contrôle. Voici la liste des segments:

- Les élément séquentiel S: ils décrivent une instruction séquentielle (par exemple: assignation, déclaration, appel de fonction ou de procédure, instruction E/S, commentaires). Les arcs qui contiennent un segment S obtiennent un poids. Les poids sont déterminés en comptant le nombre d'instructions dans l'arc. Les poids sont alors un moyen

[17] comme les boucles (do, while, repeat, until, etc), les instructions conditionnelles (if, else).

d'évaluer l'importance des décisions en terme d'instructions. Un élément S ne contribue pas au contrôle du flux.

- Les éléments label L: ils décrivent un point de référence dans le programme.
- Les éléments de branchement en avant F: ils décrivent un saut dans la direction du flux de contrôle. Un élément F est toujours connecté à un élément L.
- Les éléments de branchement en arrière B: ils décrivent un saut dans le sens inverse du flux de contrôle. Un élément B est toujours connecté à un élément L.
- Les éléments conditionnel C: ils décrivent une expression à évaluer et qui résulte en N alternatives (N>1). Par exemple une expression booléenne dans une instruction IF. Il y a toujours N éléments (Fou B) attachés à un élément C.

Ces éléments forment un ensemble complet pour décrire tout code source.

Les 4 étapes du processus de transformation du code source en graphe de contrôle sont les suivantes:

1- La traduction des instructions du code source en un graphe fait d'éléments primaires (arcs primaires).
2- La réduction des éléments qui ne contribuent pas à une information additionnelle au graphe de contrôle.
3- La concaténation de séquences d'éléments primaires en éléments secondaires, conduisant à un graphe concatené.
4- L'association des éléments primaires aux éléments secondaires, conduisant au graphe de contrôle final.

Ces étapes doivent, bien entendu, être exécutées une après l'autre, sans changer l'ordre de la transformation.
Le résultat est une représentation globale du flux de contrôle du programme.
La première étape dépend du langage de programmation. Les autres étapes sont indépendantes du langage.

5.2.2.2 Les règles de traduction.

A la première étape, un graphe avec un sommet pour chaque élément primaire est généré à partir du code source. Il faut pour cela utiliser des règles de traduction qui dépendent beaucoup du langage de programmation. C'est assez intuitif. Voici quelques règles de base:

- Chaque instruction est associée à un sommet (on introduit des numéros de ligne).
- C, L et F sont des arc en avant (ils sont représentés par une ligne verticale).
- B est un arc de retour (il est représenté par une double ligne verticale).
- Chaque instruction séquentielle produit un arc S.
- Chaque point de référence (label, début de boucle, else, endif, etc) produit un arc L.
- Chaque saut de retour inconditionnel produit un arc B (toujours connecté à un arc L).
- Chaque saut en avant inconditionnel produit un arc F (toujours connecté à un arc L).

- Chaque instruction conditionnelle (avec N alternatives) produit un arc C avec N arcs (F ou B).

Pour plus de renseignements sur les règles de traduction, il faut consulter le guide de référence du langage en question.

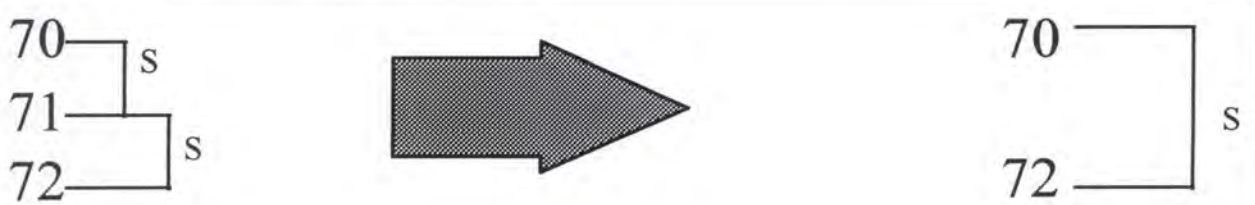
5.2.2.3 les règles de réduction.

La deuxième étape a pour but de simplifier le graphe de contrôle. Elle consiste à réduire les éléments qui ne contribuent pas au contrôle du flux dans le graphe (c'est à dire les arcs S et L). Ces règles concernent les arcs S et L.

Les règles sont les suivantes:

- Règle R1:

  Si un arc S est immédiatement suivi d'un autre arc S, on obtient alors un seul arc S.
  On a alors : S + S -> S.



Règle R1.

- Règle R2:

  S + L -> L et
  L + L -> L.

Règle R2.

5.2.2.4 Les règles de concaténation.

Indépendamment du programme, certains des éléments primaires peuvent être concaténés sans affecter la représentation du flux de contrôle du programme. Ceci est utilisé pour produire à la troisième étape des éléments secondaires dont voici la liste: SC, SF, LS, LF, LC, LSC, LSF.

Les règles de concaténation sont les suivantes:

- Règle C1:

  S + C -> SC.
  S + F -> SF.

- Règle C2:

  L + C -> LC.
  L + F -> LF.
  L + S -> LS.

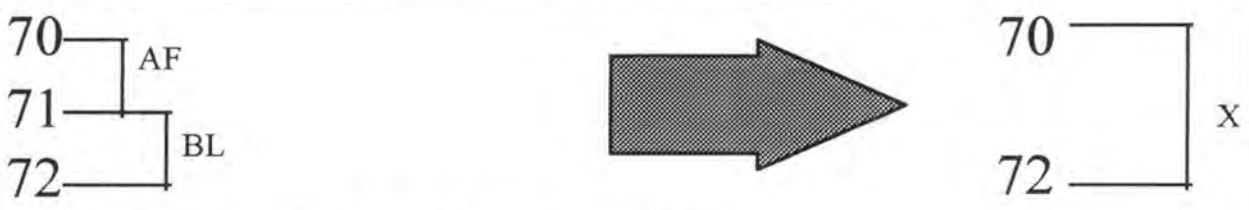
- Règle C3:

  LS + C -> LSC.
  LS + F -> LSF.

5.2.2.5 Les règles d'association.

Le graphe de contrôle peut encore être simplifié grâce aux règles d'association, utilisées à l'étape 4.

- Règle A1:



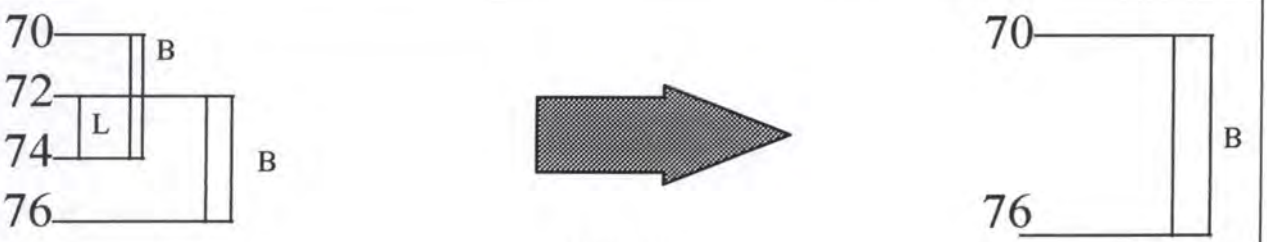
avec (A qui appartient à {S,LS} et B appartient à {F,e})
où (A appartient à {L,e} et B appartient à {S,C,F,SC,SF,e}).
x =L si a=e et b=e sinon x=ab.
e est un segment vide.

Règle A1.

On a donc:

1. S + LF -> SF.
2. SF + L -> S.
3. LSF + LF -> LSF.
4. LSF + L -> LS.
5. LF + LS -> LS.
6. LF + LC -> LC.
7. LF + LF -> LF.
8. LF + LSC -> LSC.
9. LF + LSF -> LSF.
10. LF + L ->L.
11. F + LS -> S.
12. F + LC -> C.
13. F + LF -> F.
14. F + LSC -> SC.
15. F + LSF -> SF.
16. F + L -> L.

- Règle A2:



Règle A2.

B + L + B -> B.

5.3. Les Métriques.

Les différentes métriques vont être décrites. Elles mesurent toutes des attributs internes de l'unité. Certaines métriques sont calculées directement à partir du graphe de contrôle et d'autres à partir du code source. A l'heure actuelle, on ne peut analyser qu'une unité à la fois. On ne pourra alors pas regrouper les fonctions d'un module et produire des métriques relatives au module.

Il y a 4 groupes majeurs de métriques dans Datrix. Les métriques concernant:

- la taille du code.
- le graphe de contrôle.
- la construction.
- les commentaires.

UNE REGLE GENERALE: Les métriques se rapportent à une partie de programme.
Parfois on ne voit pas de façon évidente à quelle partie se rattachent des instructions (par exemple les instructions trouvées dans un fichier "include" en C).
Datrix ne prend pas en compte les instructions qui ne font pas partie de l'unité de programme analysée. Ainsi dans l'exemple de la *figure 5.1*, Datrix ignore les instructions dans le fichier "include". Datrix attribue les commentaires avant une unité à cette unité.
Ainsi, dans le programme suivant (*figure 5.1*), le commentaire /* fin du fichier */ n'est rattaché à aucune unité.

```
#include <stdio.h>
main ()
{
printf(%,bonjour) ;
} /* fin du fichier */
```

Figure 5.1: Un programme en c.

5.3.1 Les métriques concernant la taille du code.

L'importance de la mesure de la taille d'un système a été mis en évidence en 2.1.
De nouvelle métriques de taille comme le nombre d'instructions sont introduites dans Datrix.

CS: Number of control statements (nombre des instructions de contrôle).
Cette métrique évalue le nombre d'instructions de contrôle du code source de l'unité analysée.
Plus CS est important, plus l'unité est complexe.
Domaine[18]: CS >=0.
Dans le fichier source de la *figure 5.2*, la valeur de Cs est 2. Il y a 2 instructions de contrôle:

- à la ligne 6 (le while).
- à la ligne 09 et 10 (le if else).

[18] Le domaine, comme pour une fonction, indique les valeurs possibles de la métrique.

```
01 #include <stdio.h>
02 /* ce programme compte de 90 a 10 de 10 en 10 et de 10 a 1 de 1 en 1.*/
03 main()
04 {
05 int i=90 ;
06 while (i>0)
07 {
08 printf(%d\n,i) ;
09 if (i>10) i = (i-10) ;
10 else i-- ;
11 }
12
13 }
```

figure 5.2: Un programme en C.

DS: Number of declarative statement (nombre d'instructions de déclaration).
Cette métrique évalue le nombre d'instructions de déclaration dans une unité. DS est une mesure du nombre d'opérandes η_2, introduit par Halstead (voir 2.1.2).
Domaine: DS >=0.
Pour le programme de la *figure 5.2*, la valeur de DS est 1 (ligne 05).

ES: Executable statements (instructions exécutables).
Cette métrique compte toutes les instructions exécutables dans une unité (on ne compte pas les instructions de déclaration et de contrôle). Une instruction exécutable est une portion de code où une assignation peut prendre place.
Domaine: ES >=0.

Pour le programme de la *figure 5.2*, la valeur de ES est 8.

1. ligne 06: i>10.<
2. ligne 08: printf.
3. ligne 08: %d\n
4. ligne 08: i .
5. ligne 09: i>10.
6. ligne 09: i=.
7. ligne 09: i-10.
8. ligne 10: i--.

LOC: Line of code (lignes de code).
LOC est le nombre de lignes de code dans une unité. On ne prend pas en compte les lignes vides.
Remarque: LOC ne prend pas en compte les lignes de code des fichiers "include" ou les directives au préprocesseur s'il y en a.
Domaine: LOC >=0.

Pour le programme de la *figure 5.2*, la valeur de LOC pour notre programme est de 11. Les lignes 1 et 12 ne sont pas dans l'unité.

NE: Number of exit summits (nombre de sommet de sortie).
NE est le nombre de sommets dans le graphe de contrôle où le flux de contrôle s'arrête ou retourne à l'unité appelante.
Remarque: les sommets de sortie sont représentés par le symbole: <- dans de graphe de contrôle.
Une unité peut avoir plus d'un point de sortie.
Cette métrique peut être utilisée pour détecter les points de sortie multiples, qui sont indésirables.
Domaine: NE >=1.
Pour le programme de la *figure 5.2*, NE =1.

NI: Number of entry summits (nombre de sommets d'entrée).
Remarque: Les sommets d'entrée sont représentés par le symbole: ->.
Cette métrique peut être utilisée pour détecter les points d'entrée multiples, qui sont indésirables.
domaine: NI>=1.

S: Number of statements (nombre d'instructions).
S est le nombre de toutes les instructions (déclaratives et les instructions de structure) dans une unité.
S = CS + DS + ES.
Domaine: S >=0.
Pour le programme de la *figure 5.2*, S=Ds + Es + Es = 2 + 1+ 8 =11.

5.3.2 Les Métriques concernant le graphe de contrôle.

Les métriques présentées dans cette section donnent des mesures de la structure logique.

BS: Number of Branches of Structure (nombre de branches de structure).
C'est le nombre d'arc croisant le graphe de contrôle, ce qui viole les principes de la programmation structurée (c'est à dire l'usage de structures de contrôle avec une entrée et une sortie).
Remarque: les branches sont représentées par un carré noir dans le graphe de contrôle. Quand Datrix vérifie un croisement pour voir si c'est une branche de structure, il simplifie les instructions de l'arc là où est le croisement (selon les règles de réduction, concaténation et association). Si les instructions de l'arc respecte les principes de la programmation structurée, Datrix va simplifier les instructions et déterminer que le croisement n'est pas une branche de structure. Si Datrix ne peut pas simplifier ces instructions, alors le croisement est une branche de structure.
Si une fonction est trop longue ou trop complexe, il se peut que Datrix ne soit pas capable d'évaluer si un croisement est une branche de structure ou non (il y a une limite de temps pour cette métrique).
La Valeur de BS pour un programme structuré est 0.
Domaine: BS >= 0.

Pour le graphe de la *figure 5.3*, BS=1 (ligne 75).

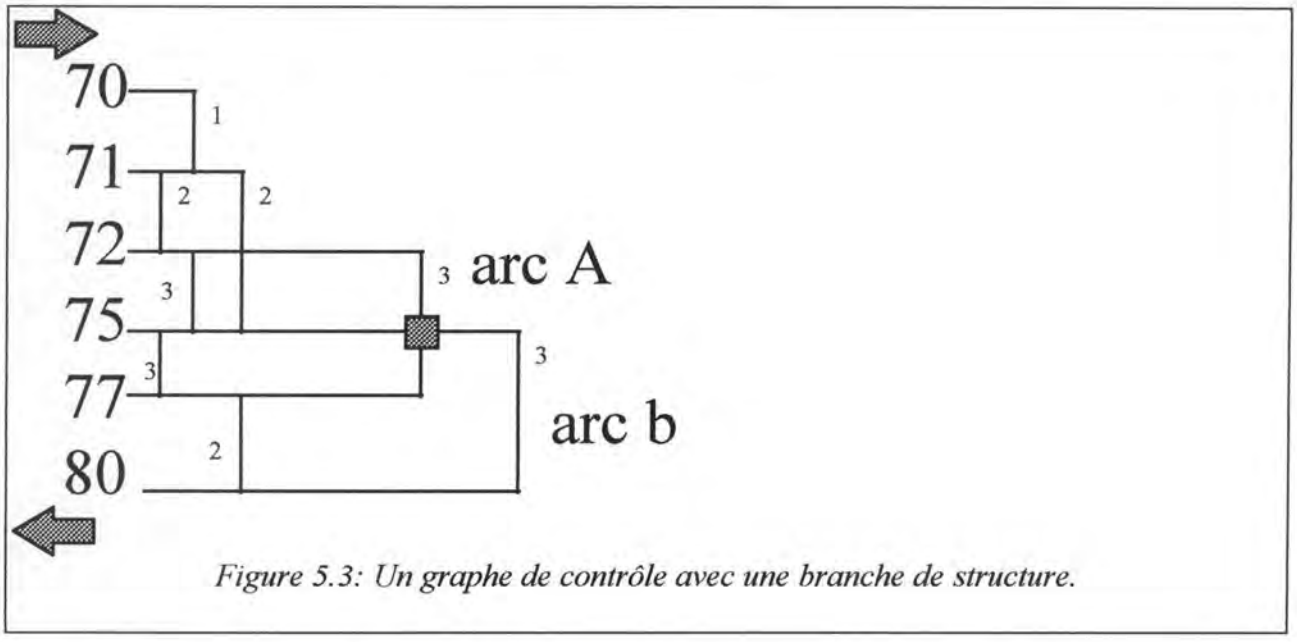


Figure 5.3: Un graphe de contrôle avec une branche de structure.

BSP: Weighted number of branches of structure.
BSP est la somme des poids de tous les arcs impliqués dans les branches de structure.
Quand il y a une branche de structure, les arcs dans la portée des 2 causant la branche sont examinés et leurs poids sont ajoutés. BSP donne une idée du nombre d'instructions comprises dans la section violant les principes de la programmation structurée.
Remarque: la valeur de BSP pour un programme structuré est 0. Les poids des arcs respectifs sont assignés pour chaque langage par l'analyseur.
Domaine: BSP >=0.
Pour le graphe de la *figure 5.3*, les arcs en question sont A et B. Il y a un arc entre A et B: 75-77. Son poids est de 3, donc BSP=3.

E: Number of arcs (nombre d'arcs).
C'est le nombre total d'arcs trouvés dans le graphe de contrôle. Le calcul est fait en comptant chaque ligne verticale dans le graphe. E est utilisé pour calculer d'autres métriques comme VG.
Domaine: E >=1.
Pour le graphe de la *figure 5.3*, E=8.

K: Number of knots (Nombre de croisements).
C'est le nombre de croisements dans le graphe de contrôle. Cela inclut les croisements qui violent les principes de la structuration programmée et ceux qui ne les violent pas.
Domaine: K >=0.
Pour le graphe de la *figure 5.3*, K=2: il y a des croisements à la ligne 72 et à la ligne 75.

V: Number of summits (nombre de sommets).
C'est le nombre de sommets dans le graphe de contrôle. Les sommets sont représentés par des lignes horizontales dans le graphe de contrôle. V est utilisé pour calculer d'autres métriques comme VG.
Remarque: il y a au moins 2 sommets dans un graphe de contrôle.
Domaine: V >=2.
Pour le graphe de la *figure 5.3*, V=6.

VG: Cyclomatic number (nombre cyclomatic).
VG = E - V + NE + NI.
VG est évalué en prenant le nombre d'arcs (E) moins le nombre de sommets (V) et en ajoutant le nombre de sommets d'entrée (NI) et de sortie (NE).
Domaine: VG>=1.
Cette métrique donne une idée du nombre de chemins indépendants que l'on peut suivre, ce qui donnera des indices pour le nombre de test que l'on peut faire.
Pour le graphe de la *figure 5.3*, VG = 8-6+1+1 = 4.

5.3.3 Les métriques concernant la construction.

Là encore, les métriques présentées dans cette section donnent des mesures de la structure de l'unité.

NC: Mean conditionnale arc complixity.
NC donne une idée de la complexité moyenne pour tous les arcs conditionnels. Une grande valeur de NC indique que les conditionnelles sont difficiles à évaluer et dépendantes de beaucoup d'opérateurs et de variables. Afin d'évaluer la complexité des conditionnelles, les valeurs de complexité suivantes sont appliquées au contenu des expressions booléennes:

- constantes: 1.
- opérateur: 1.
- identifiant: 2.
- autre: 0.

Ce qui constitue les constantes, les opérateurs et les identifiants dépend de chaque langage. En conséquence, le calcul des métriques dépend d'un langage à l'autre.
Domaine: NC >=0.
Pour le programme de la *figure 5.2*, NC = (4+4) / 2 = 4. On a 2 expressions booléennes dans des arcs conditionnels:
ligne 06: i>0.
ligne 09: i>10.

NCMAX: Maximal conditionnal arc complexity.
NCMAX est la valeur maximale de complexité de tous les arcs conditionnels. Cette variable est comparée avec NC pour donner une vue globale de la complexité des conditionnelles, en observant la variance par exemple.
Domaine: NCMAX >=0.
Pour l'exemple de *la figure 5.2*, NCMAX = 4.

NCN: Number of conditionnal arcs.
NCN est le nombre des arcs conditionnels dans le graphe de contrôle.
Remarque: un arc conditionnel est un arc C.
NCN permet d'évaluer les différents chemins qui peuvent être exécutés dans l'unité. Cela donne une idée du nombre de décisions faites dans l'unité.
Domaine: NCN >=0.
Pour le graphe de la *figure 5.3*, NCN= 3 (arc 70-71, arc 71-72, arc 72-75).

NEL: Mean nesting level.
NEL est le niveau moyen de nichage de tous les arcs. Cette métrique représente l'utilisation des structures de contrôles nichées.
Domaine: NEL >=1.
Pour le graphe de la *figure 5.3*, le niveau de nichage est indiqué sur chaque arc (Par exemple, comme il y a 2 arcs qui partent du sommet 72, ils ont tous les 2 un niveau de nichage de 2).
NEL = (1+2+2+3+3+3+3+2)/8=19/8.

NELMAX: maximal nesting level.
NELMAX est la valeur maximale de niveau de nichage dans toute l'unité. NELMAX peut être comparer à NEL pour avoir une vue globale des niveaux de complexité de nichage.
Domaine: NELMAX >=1.

NELW: Weighted mean nesting level.
NELW est la moyenne du niveau de nichage, en tenant compte des poids des arcs.
Remarque: NELW représente la distribution des instructions exécutables dans les niveaux de nichage.
Une grande valeur de NELW suggère que les instructions exécutables sont plus concentrées dans les arcs avec des poids élevés.
Domaine: NELW>0.
Pour le graphe de la *figure 5.3*, supposons que le poids des arcs soit identique à leur nichage.
NELW = (1*1 + 2*2 + 2*2 + 3*3 + 3*3 + 3*3 + 3*3 + 2*2)/(1+2+2+3+3+3+3+2) = 49/19.

NL: Number of loop constructs.
C'est le nombre d'arcs de retour dans le graphe de contrôle. On a des boucles lorsqu'on utilise les instructions Do.. While, Repeat .. Until , etc.
Domaine: NL >=0.
Pour le graphe de la *figure 5.4*, NL =1.

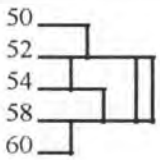


Figure 5.4: Un graphe de contrôle.

NP: Number of independant paths.
NP est le nombre de chemin que le flux de contrôle peut suivre depuis le(s) sommet(s) d'entrée jusqu'au(x) sommet(s) de sortie. Cela représente chaque possibilité que l'on a dans l'unité pour entrer et sortir, suivant l'exécution du programme.
Domaine: NP >=1.
Pour le graphe de la *figure 5.3*, voici quelques chemins différents:

70-71-72-75-77-80
70-71-75-77-80.
70-71-72-77-80
ect...

PSC: Mean conditionnal arc span.
PSC est l'étendue moyenne des branches des arcs conditionnels. Cette métrique est exprimée en nombre d'arcs unitaires. Un arc unitaire est un arc qui est localisé entre 2 sommets consécutifs.

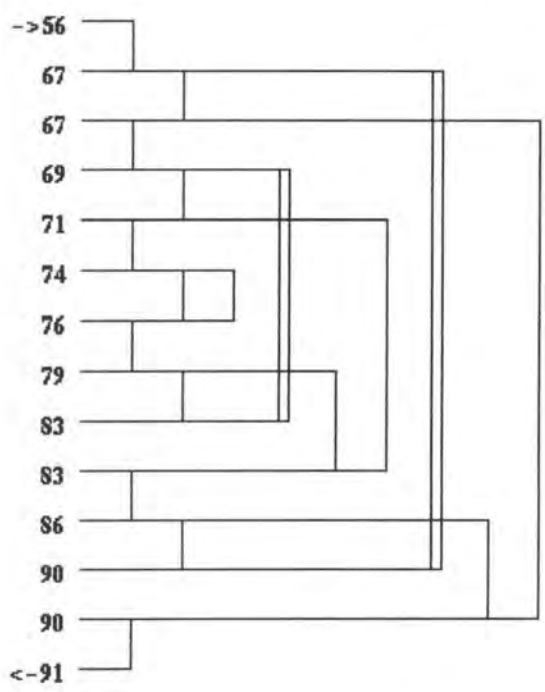


Figure 5.5: Un graphe de contrôle.

Pour le graphe de la *figure 5.5*, Psc =4.

L'arc 67-67 est un arc conditionnel et a une branche d'étendue 10.
L'arc 69-71 a une branche d'étendue de 5 .
L'arc 71-74 a une branche d'étendue 1.
L'arc 76-79 a une branche d'étendue 2, de même que l'arc 83-89.

PSCMAX: Maximum conditionnal arc span.

PSCMAX est le nombre maximal d'arc unitaires qui sont localisés entre l'étendue d'une branche d'un arc conditionnel.

On peut comparer PSCMAX à PSC pour avoir une vue globale de l'étendue des arcs conditionnels.

Domaine: PSCMAX >=0.

Pour le graphe de la *figure 5.5*, PSCMAX = 10.

VS: Structural volume.

VS = (E + V + 2)/5.

VS est calculé en ajoutant le nombre des arcs E et le nombre des sommets V plus 2. Le total est divisé par 5. Chaque conditionnel ou boucle doit augmenter le volume structurel par 1.

Remarque: pourquoi 2 et 5?

La valeur de Vs doit être de 1 pour une unité minimale, quand on a un sommet d'entrée, un sommet de sortie et un arc. Dans ce cas, E = 1 et V = 2.

La valeur de VS doit être de 2 pour une unité minimale avec une conditionnelle, comme pour le graphe de la *figure 5.6*, où E = 4 et V = 4:

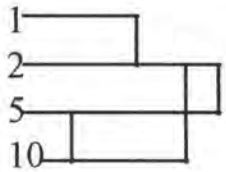


Figure 5.6: Un graphe de contrôle.

On a 2 équations à 2 inconnues:

(E+V+X)/Y=1 => (3+X)/Y=1.
(E+V+X)/Y=2 => (8+X)/Y=2.

La solution est: X=2 et Y=5.
Domaine: Vs >=1.
Pour le graphe de la *figure 5.3*, VS = (8+6+2)/5=16/5.

RLS: Loop structural ratio.
RLS = 1/Vs * Σ (Eloop + Vloop + 2)/5.
Vloop est le nombre de sommets localisés dans l'étendue d'un arc de retour.
Eloop est le nombre d'arcs dans l'étendue du même arc de retour, incluant les arcs connectés au début et à la fin de l'arc de retour. RLS est le ratio du volume structurel contenu dans les sous graphes délimités par des arcs de retour (boucles) sur le volume total structurel.
RLS ne peut être plus grand que 1 si les boucles sont nichées.
Domaine: RLS >=0.
Pour le graphe de la *figure 5.5*, RLS = 1.49, VS = 6.80.
Il y a 2 boucles:
Pour la boucle 83-69, Eloop=11, Vloop=6.
Pour la boucle 90-67, Eloop=19, Vloop=11.
Donc RLS= (1/6.80)*((11+6+2)+(19+11+2))/5=1,49.

VW: Sum of weights.
VW est la somme des poids de tous les arcs dans le graphe de contrôle.
Domaine: VW >=0.
Pour le graphe de la *figure 5.3*, VW = (1+2+2+3+3+3+3+2)=19.

RLW: Loop weight ratio.
RLW = (1/VW)*Σ(We) où Σ(We) est la somme des poids de tous les arcs connectés aux sommets localisés dans l'étendue de l'arc de retour.
RLW est le ratio de la somme des poids dans les sous graphes délimités par les arcs de retours (loop) sur la somme totale des poids. RLW ne peut être plus grand que 1 si les boucles sont nichées.
Domaine: RLW >=0.

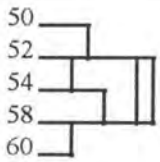


Figure 5.7: Un graphe de contrôle.

Pour le graphe de la *figure 5.7*, on a les poids suivants: poids(50-52)=1, poids(52-54)=2, poids (54-58)=3, poids (58-60)=1.
VW=1+2+3+1=7. RLW = (3+2)/7= 5/7.

VP: Number of pending summits (Nombre de sommets pendants).
C'est le nombre de sommets dans le graphe de contrôle qui n'ont pas de sommets d'entrée. Tous les arcs qui suivent immédiatement un sommet pendant correspondent à du code mort (du code qui ne sera jamais exécuté). Un sommet pendant est un chemin inaccessible dans une unité, comme on le voit sur la *figure 5.8*. VP=1. Le sommet 54 est inaccessible.
Domaine: VP<=0.

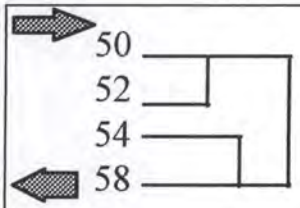


Figure 5.8: Un graphe de contrôle.

5.3.4 Les métriques relatifs aux commentaires.

Datrix attache une grande importance à évaluer la documentation du code.

C: Number of logical comments (Nombre de commentaires logiques).
Cette métrique évalue le nombre de commentaires logiques dans une unité. Un commentaire logique est défini comme un ensemble de commentaires qui ne sont pas séparés par des instructions.
Domaine: C >=0.

```
01 #include <stdio.h>
02 /* ce programme compte de 90 a 10 de 10 en 10 et de 10 a 1 de 1 en 1.*/
03 /* voici la fonction principale */
 main()
04 {
05 int increment = 0 , unused, i=90 ; /* unused n'est pas utilise */
06 while (i>0) /* boucle jusqu'a ce que i vaille 1*/
07 {
08 increment++ ;
09 printf(%d\n,i) ;
10 if (i>10) i = (i-10) ;
11 else i-- ;
12 }
13
14 }
```

figure 5.9: Un programme en C.

Pour le programme de la *figure 5.9*, C = 2. Il y a un commentaire logique aux lignes 02, 03, et un autre à la ligne 06. On peut utiliser cette métrique pour voir si le nombre de commentaires est satisfaisant. C ne fournit pas pour autant une mesure de la qualité des commentaires.

LS: Mean length of variable names (longueur moyenne des noms de variables).
LS est le nombre moyen de caractères de toutes les variables qui sont utilisées dans une unité. Les variables déclarées non utilisées ne rentrent pas en compte. Une grande valeur de LS indique que les noms de variables sont longs et donc plus représentatifs, par hypothèse. Bien sûr, il faut admettre que les noms de variables sont significatifs.
Domaine: LS >0.
Pour le programme de la *figure 5.9*, LS = (9+1) / 2=5. La variable i n'a rien de significatif et est courte, faisant chuter la moyenne.

RAC: Code arc comments ratio.
RAC est le pourcentage des arcs commentés (arcs composés au moins d'un segment S) sur le nombre total de S arcs.
RAC = 100 * Ac/A où Ac est le nombre de S arcs commentés et A est le nombre total de S arcs. La valeur de RAC est exprimée en pourcentage. Les arcs qui ne sont pas vides doivent être composés au moins d'un segment S, LS, LSF sont des exemples d'arcs qui ne sont pas vides.
Datrix attache toujours les commentaires trouvés à la prochaine instruction exécutable dans le code source.
Domaine: 0<= RAC <= 100.
Pour le programme de la *figure 5.9*, si on suppose qu'il y ait 1 arc S commenté sur un total de 5 arcs S, RAC = 20%.

RCC: Commented control structure ratio.
RCC est le pourcentage de structures de contrôle commentées par rapport au nombre total d'arc de structure de contrôle.
RCC = 100* Ac/A où Ac est le nombre d'arc de structures de contrôles commentés (on n'inclut pas les arcs S) et A est le nombre total d'arcs (en n'incluant pas les arcs S).
Remarque: cette métrique est utilisée pour mesurer combien de commentaires sont présents dans les structures de contrôles. L'arc S n'est pas pris en compte parce que ce n'est pas un arc de structure de contrôle.
Domaine: RCC >=0.
Pour le programme de la *figure 5.9*, si on suppose que dans un graphe il y 1 arc L commenté et en tout 4 arcs non S, RCC = 25%.

VCS: Comments Volume in Structures.
VCS est le nombre total de caractères alphanumérique trouvés dans les commentaires localisés partout dans l'unité excepté dans la section déclaration. Cette métrique est complémentaire au volume de commentaires dans la partie déclaration (VCD). Aucune vérification n'est faite pour s'assurer qu'un commentaire a au moins 12 caractères alphanumériques. Cette métrique ne fait que compter le nombre de caractères.
Domaine: VCS >=0.
Pour le programme de la *figure 5.9*, VCS=26.

RVC: Comments volume ratio.
RVC = VCS/VS.
RVC est le ratio entre le volume de commentaires dans la structure (VCS) et le volume de structure (VS).
Domaine: RVC>=0.

VCD: Comments Volume in Declarations.
VCD est le nombre total de caractères alphanumériques trouvés dans les commentaires localisés dans la section déclaration. Seuls les caractères alphanumériques sont comptés.
Domaine: VCD >=0.
Pour le programme de la *figure 5.9*, VCD = 20. Les ":" et les "." ne sont pas comptés.

5.4 Critique de Datrix.

Datrix fournit des métriques d'attributs internes, qui sont très utiles pour évaluer la qualité d'un logiciel[19].
Les métriques de taille de Datrix mesurent plus que la longueur (excepté pour LOC) et sont basées pour la plupart sur le nombre d'instructions (ce qui donne une idée de la complexité). Datrix ne permet malheureusement pas de prédire la longueur à partir du design.

[19] voir le chapitre 2.

Les métriques relatives au graphe de contrôle et à la construction du programme donne une bonne indication de la structure. Mais il n'est pas toujours évident de tirer des conclusions objectives à partir de métriques conceptuellement complexes comme RLS (le ratio du volume structurel contenu dans les sous graphes délimités par des arcs de retour (boucles) sur le volume total structurel).

Les métriques relatives à la documentation ne donnent aucune indication de la qualité des commentaires (qui ne dépend pas de la longueur). Datrix permettra d'attirer l'attention sur un manque potentiel de commentaires et l'utilisateur pourra ensuite vérifier la documentation du code.

L'utilisation de Datrix n'est pas triviale et requiert une formation. En général, il est difficile d'inférer des critiques uniquement à partir des métriques (sauf pour quelques métriques comme Vp, le nombre de sommets pendants. Il est alors évident que la structure du code est a modifier puisque certaines instructions sont inaccessibles). Une inspection du code s'avère alors nécessaire pour juger de la qualité du logiciel.

CHAPITRE 6:

AMÉLIORATION DE LA QUALITÉ PAR DES PRATIQUES DE DESIGN ET DE PROGRAMMATION, DES OUTILS ET DES TECHNIQUES.

Pour obtenir des logiciels de qualité, cela ne suffit pas de mesurer des attributs à l'aide de métriques. Il faut aussi améliorer le processus de développement. Des méthodes, des outils et des techniques peuvent être utilisés pour cela.
Des pratiques de design et de programmation seront présentées en 6.1 et 6.2 respectivement. Les section 6.3 et 6.4 concerneront des techniques et outils utilisés en Software Quality Assurance[20] pour améliorer la qualité des logiciels.

6.1 Pratiques de Design pour Améliorer la Qualité.

Lawrence J.Peters, dans "Comparing Software Design méthodologies", a publié des pratiques et des méthodes de design.

Le design est une phase particulièrement importante du développement. Il a été constaté que beaucoup de fautes étaient dues à un mauvais design. Généralement ces fautes sont découvertes tardivement dans le cycle de développement, aux moments des tests. Trouver et corriger un problème après la livraison du produit coûte 100 fois plus cher que de le faire durant le design. Il faut alors se concentrer sur le design, qui est une phase difficile étant donné que les besoins sont souvent mal compris.

Le section 6.1.1 montrera l'impact du design sur la qualité. En 6.1.2, les obstacles à un bon design seront présentées et en 6.1.3 nous esquisseront quelques pistes pour réaliser un design de qualité.

6.1.1 Impact du design sur la qualité.

Une bonne découpe en modules durant le design permettra d'avoir un logiciel facile à étendre et à maintenir. Lors de l'implementation, il ne restera pas de fautes majeures (les fautes dues à un mauvais design sont les plus difficiles à corriger. Elles peuvent entraîner beaucoup de modifications dans d'autres modules autres que celui où est apparue la faute). La reliability sera

[20] voir chapitre 7.

alors améliorée. Il apparaît aussi qu'un logiciel efficace (qui produit le moins d'objets possibles) dépend d'une bonne structure de données et donc d'un bon design.
Il est important d'avoir un design de qualité, car même si cela ne l'a pas prouvé, il est évident qu'un bon design conduira à un logiciel ayant les 4 caractéristiques suivantes: maintenabilité, efficacité, extensibilité et reliability.

6.1.2 Obstacles à « un design de qualité ».

Peters identifie 3 obstacles:

- Les niveaux de design:
  Généralement, on peut voir l'aboutissement du design à 2 niveaux: l'architecture globale du système ou, à un niveau plus bas, la structure et les opérations d'éléments individuels du design. Si on ne voit le design qu'à un seul niveau, on risque d'aboutir à un système avec une bonne architecture mais avec des composantes individuelles pauvres ou l'inverse. Il apparaît qu'il est plus facile de traiter à un niveau détaillé et localisé qu'au niveau du système: il est plus facile de discuter des formats d'écran pour un système graphique que de la philosophie sur laquelle est basée l'architecture du système. On retiendra ici qu'il faut s'efforcer de voir le design à différents niveaux.

- Les spécifications instables:
  C'est le problème le plus critique dans le design. Dans la phase de design, on peut être amené à modifier les spécifications, d'une part parce que le problème a changé et d'autres part parce que le designer comprend mieux le problème. La phase de design est alors déstabilisée et il est facile de prédire que la qualité en pâtira. Il faut remarquer que les liens entre les éléments du problème réduisent la facilité de gérer d'une part les définitions des spécifications et d'autre part les changements. Par exemple, si on a des spécifications au niveau de la taille du programme et au niveau du temps d'exécution, il sera difficile de modifier une spécifications sans modifier l'autre.

- L'approche inflexible:
  Par la pression des managers, dans beaucoup d'organisations, on a adopté des méthodes de design fixes qui ne sont pas toujours adaptés aux problèmes de design rencontrés.

6.1.3 Une approche pour réaliser un design de qualité.

Il faut bien garder en tête qu'un design est spécifique à un problème. Ce qui suit ne sont pas des règles à adopter à tout prix:

- L'intégrité du design:

On doit relier les éléments du système via un thème général, donc définir prudemment son architecture, faire des reviews. Même si ce thème n'est pas toujours évident, grâce aux reviews[21], il doit apparaître. On doit alors pouvoir le justifier. Un exemple de justification serait: nous avons suivi cette approche car elle maximise l'utilisation d'une application existante.

- Un design rationnel :

Peters considère qu'on a publié beaucoup d'approches pour le design de logiciel mais relativement peu d'approches incluent une base rationnelle sur les lignes à suivre pour satisfaire des besoins détaillés.

Une approche rationnelle indiquera, par exemple:

1. La structure de données est la clé d'un design effectif. (Jakson: "principles of program design", 1975).
2. Le flux de données est la clé d'un design réussi. (Youdon, Constantine: "structured design", 1975).

Toutes les approches publiées incluent des techniques utiles comme:

1. Redéfinir le design itérativement, pas à pas.
2. Utiliser des graphiques spécialisés comme des bubble chart, data structure diagramme, et des diagrammes de flux spécialisés.

Une méthode est plus utile qu'une technique car pour les problèmes complexes où la solution n'est pas évidente, une technique seule ne guidera pas. Une méthode est plus efficace car elle capture le "what" et en principe le "how" (pas à un niveau détaillé, ce qui est laissé au designer).

- Connaître son utilisateur:

Face au problème de l'utilisateur qui rejette le système, il est important de développer des modèles utilisateurs, comme des diagrammes de flux qui lui permettent de comprendre de manière rudimentaire comment le système fonctionne.

- Faire correspondre le problème et la solution:

Il faut bien garder en tête que chaque méthode et technique est relative à un type de problème. Une méthode est adaptée bien souvent pour rencontrer les caractéristiques du problème, les contraintes au niveau des ressources, du client, du calendrier, des coûts.

Voici des exemples de méthodes (dans les grandes lignes):

1. Le design structuré: cela consiste à utiliser des charts de structure pour représenter l'architecture. La solution est développée itérativement.
2. La méthodologie de Jackson: cela consiste à utiliser des charts comme les arbres pour représenter la structure de données et à définir des lignes à suivre.

- Définir des objectifs de design:

En effet, il faut fixer les points importants à respecter de manière concise, mesurable et disciplinée (ce qui n'est pas toujours le cas). Sans objectif, la qualité du design peut être dégradée.

---

[21] ce terme est défini en 6.3.

Un objectif est un but à atteindre qui a au moins les éléments suivants:

1. Un définition du problème. Par exemple: le temps de réponse est trop élevé de 2 secondes.
2. Une définition de l'objectif, de manière mesurable. Par exemple: réduire le temps de réponse de 2 secondes.
3. Définir comment mesurer l'objectif. Pour notre exemple, cela consiste à définir le temps de réponse comme étant le temps entre la pression du bouton "enter" et l'apparition à l'écran d'une réponse complète.

En conclusion, on peut dire que même si la qualité à des variables non mesurable, les pratiques de design doivent permettre d'atteindre la qualité requise et de la quantifier. Il n'y a pas de technique unique.

6.2 Des Pratiques de "Programmation" pour Augmenter la Qualité des Logiciels.

Historiquement, c'est l'Air Force qui a joué une part importante dans l'établissement de beaucoup de pratiques de programmation modernes. Les chercheurs de l'Air Force font l'hypothèse suivante: " La production de logiciel de meilleure qualité et à moindre coût est rendue possible par l'utilisation de règles de développement de logiciel définies rigoureusement, supportées par des outils et des techniques modernes."

Vers 1977, RADC (Air Force at Rome Air Developpement Center) a voulu étudier cet impact qui a été montré comme positif (Brown, "Impact of MPP on System Development", 1977).

L'étude a révélé que les pratiques de programmation modernes (MPP) qui ont un impact le plus élevé sont les suivantes:

- Les spécifications formelles et l'analyse des besoins: la simulation peut être utilisée pour montrer que les besoins de performance sont raisonnables. Les reviews des spécifications permettent de résoudre les problèmes.

- Une méthodologie de design rigoureux, Top-down: c'est une approche où, partant d'une formulation du problème au plus haut niveau, on décompose en fonctions par itérations successives. Des techniques comme SADT, les charts hiérarchiques sont utilisées.

Les MPP avec un impact encore plus élevé sont:

- L'approche par phase, production incrémentale (comme le modèle Waterfall): pour les grands projets, le processus de développement est vu comme une séquence de phases disctinctes mais très liées. Par exemple, la définition des besoins, un design préliminaire, un design détaillé, une implémentation, les tests, la maintenance. Il peut y avoir itération.

- La documentation in process: cela consiste à générer, maintenir et revoir un document que l'on complète incrémentalement et qui contient toute l'information pertinente sur le développement et les tests (besoins, données de design, code, diagramme de flux, plans pour les tests, résultat des tests). Le plan de développement est détaillé. Des dates sont assignées pour la complétude des différentes étapes.

- Les outils de développement et de test: comme les simulateurs, les flowcharters.

Les MPP avec un impact encore plus élevé sont:

- Les tests indépendants: cela consiste à former une équipe de spécialistes, distincte du groupe responsable du code, du design. Les activités de ce groupe sont: l'analyse des besoins, le planning, la préparation et l'exécution des tests, la revue des tests.

- Les standards de programmation: cela consiste à établir et à suivre des standards de programmation pour produire le code et la documentation. Les pratiques typiques sont l'utilisation de commentaires, les conventions de noms ou la limitation de l'utilisation de certaines instructions de contrôle comme les "goto".

6.3 Les Techniques.

Il existe un grand nombre de techniques. Donald J.Reifer en définit 20. Sont reprises ici celles qui semblent les plus importantes:

- L'Analyse de l'Exécution: c'est une technique employée durant les tests pour mettre en évidence les erreurs du programme. Des statistiques sont établies à partir de l'exécution du programme.

- L'Auditing: c'est une technique formelle employée pour examiner et vérifier le status d'un programme et de sa documentation ou le respect de procédures établies par le personnel du projet. On s'attache ici à vérifier le respect des spécifications.

- L'Inspection du Code: c'est une technique utilisée afin d'identifier les erreurs. Des critères sont employés pour évaluer le code. Il faut s'assurer que le code est corrigé en cas d'erreur.

- L'Inspection du Design: c'est une technique utilisée afin d'identifier les erreurs. Les participants ont un rôle bien défini et des critères pour évaluer le design.

- Le Modeling Analytique: c'est une technique utilisée pour exprimer mathématiquement une représentation de problèmes réels. De tels modèles doivent représenter de manière abstraite l'essence du sujet. Si le système est trop compliqué, on pourra être amener à faire des simplifications, les équations décrivant le système n'étant pas formulables. On perd alors en précision.

- Les Preuves d'Exactitude: c'est une technique utilisée pour prouver l'exactitude du programme en utilisant des moyens similaires à ceux employés pour prouver les théorèmes mathématiques.

- Le Reviewing: c'est une technique formelle employée pour examiner et vérifier le status d'un programme et de sa documentation ou le respect de procédures établies par le personnel du projet. On s'attache ici à vérifier en détail l'adéquation technique du produit.
  Une attention particulière est portée pour détecter les bugs générés dans la phase courante car plus une erreur est détectée tôt, plus elle est facile à corriger. Les reviews sont en général jugés trop tardifs, trop limités et pas assez rigoureux (il en est de même pour l'inspection du code source et le walk-throught).

- La Simulation: en utilisant des modèles, ce processus d'attache à vérifier des alternatives sur le timing, les capacités du système ou encore la performance.

- Le Test d'Évaluation d'Algorithme: c'est une technique utilisée pour évaluer les aspects critiques de l'algorithme clé avant de terminer le design. Par exemple, la vitesse versus la taille. L'algorithme est lancé dans un environnement simulé pour s'assurer que les besoins sont satisfaits.

- Le Walk-Throught: c'est une technique utilisée pour revoir le design ou le code et identifier les erreurs. Le programmeur responsable discute de son produit avec son équipe et sollicite des avis. Les walks-throught peuvent permettre de détecter jusqu'à 60% des erreurs.

Toutes ses différentes techniques ont une efficacité variable pour vérifier les différentes caractéristiques de la qualité (voir *tableau 6.1*). De plus, elles ne supportent pas nécessairement toutes les fonctions de Software Quality Assurance (SQA), comme en témoigne le *tableau 6.2*.

Caractéristiques de Qualité						
Techniques	Efficacité	Maintenabilité	Portabilité	Reliability	Testability	Usability
Analyse de l'exécution	H	M	F	M	F	F
Auditing	M	H	M	H	H	H
Inspection du code	F	M	M	H	M	M
Inspection du design	H	H	H	H	H	H
Modeling analytique	M	F	F	H	M	F
Reviewing	M	H	M	H	H	H
Test d'évaluation d'algorithme	M	F	F	H	M	M
Walk-throught	M	M	M	H	M	M

Légende: H-Très Efficace, M- Efficacité Moyenne, F- Faible Efficacité.

Tableau 6.1: Efficacité des techniques pour vérifier les caractéristiques de qualité.

Fonctions de SQA					
Techniques	Planning de la qualité	Tests	Standards de design	Standards de documentation	Reviews et audits
Analyse de l'exécution		×			×
Auditing	×	×	×	×	×
Inspection du code			×	×	×
Inspection du design			×	×	×
Modeling analytique		×			
Reviewing	×	×	×	×	×
Test d'évaluation d'algorithme		×			
Walk-throught			×	×	×

Tableau 6.2: Comment les techniques supportent les fonctions de SQA.

Ainsi une analyse de l'exécution sera peu efficace pour vérifier la portabilité mais très utile pour vérifier l'efficacité. Il apparaît que l'inspection du code est très efficace pour contrôler la qualité en général. L'auditing, le reviewing, le walk-throught et l'analyse de l'exécution sont également très utiles pour contrôler la qualité (ce sont d'ailleurs des techniques très utiles en SQA).
Le *tableau 6.2* montre que les test sont supportés par beaucoup de techniques et que l'auditing et le reviewing supportent toutes les fonctions de SQA.

6.4 Les Outils.

Donald J. Reifer en définit 32. Les plus intéressants sont les suivants:

- L'Analyseur de B.D.: c'est un programme qui rapporte de l'information sur chaque usage de données, identifie chaque programme qui utilise la donnée, indique si le programme prend la donnée en input, l'utilise, la modifie ou la rend en input. Les données non utilisées sont imprimées. Les erreurs dans l'usage de données, les conflits sont identifiés.

- L'Analyseur Dynamique: c'est un programme qui ajoute des compteurs ou autre détecteur statistique au code et produit des rapports sur l'utilisation des différentes parties du code. On l'utilise pour le réglage, l'optimisation.

- L'Analyseur de Standards: c'est un programme utilisé pour déterminer si les standards prescrits ont été suivis. Le programme peut vérifier les conventions sur la taille du programme, les commentaires, la structure.

- L'Analyseur Statistique: c'est un programme qui examine le code source statistiquement (c'est à dire sans l'exécuter). Il analyse entre autre la syntaxe, la séquence d'événements, vérifie la structure, l'interface des modules.

- L'Analyseur de Temps: c'est un programme qui rapporte le temps d'exécution de tous les éléments du programme: fonctions, procédures, etc.

- Le Debugger: il fournit des informations comme le temps de l'exécution du programme, l'environnement de la machine, les zones mémoires utilisées.

- Le Driver de Test: pour lancer des tests, il est souvent nécessaire de décrire une situation dynamique. Le driver de test permet de générer des données représentant la situation. Ces données servent ensuite d'inputs pour le système.

- L'Editeur: c'est un programme qui analyse le code source pour détecter les erreurs comme la violation des pratiques et standards de programmation. L'éditeur construit une liste de référence des labels, variables, constantes.

- L'Éditeur de texte: il est utilisé pour faire la documentation.

- Le Flowcharter: c'est un programme utilisé pour montrer en détail la structure logique du code. Le flux est déterminé par des instructions exécutables. Autoflow et Flowgen sont des flowcharters actuellement commercialisés.

- La Librairie: c'est une collection organisée d'informations. Il y a différents types de librairies: certaines gèrent le stockage du programme, d'autres gèrent le programme, sa documentation et les données relatives aux tests.

- Le Moniteur de Logiciel: c'est un programme qui fournit des statistiques détaillées sur la performance du système. Il examine les longueurs des queues, l'usage de la mémoire, etc.

- L'Éditeur de Texte: on l'utilise pour préparer la documentation.

- Le Processeur de Langage: compilateurs, assembleur sont des exemples.

- Le Simulateur: c'est un programme qui doit fournir des données qui sont proches de celles qui seraient fournies par le périphérique simulé.

- Les Standards: ce sont des procédures, des règles que l'on prescrit durant le développement du logiciel. Des règles sur l'architecture, des conventions de langage ou de documentation sont des exemples typiques.

- La Table de Décision: cet outil permet de montrer de manière claire dans quelles conditions on décide d'une action. L'information sur les conditions, règles et actions est représentée sous forme tabulaire qui peut être automatiquement traduite en code exécutable par un processeur.

- Le Test Bed: c'est un site de test composé du hardware actuel ou d'un équipement qui simule le hardware sur lequel tournera l'application.

Comme pour les techniques, l'impact d'un outil n'est pas le même pour vérifier toutes les caractéristiques de qualité (voir *tableau 6.3*). Le *tableau 6.4* montre les fonctions de SQA supportées par chacun des outils présentés.

	Caractéristiques de la qualité					
Outils	Efficacité	Maintenabilité	Portabilité	Reliability	Testabilité	Usability
Analyseur de B.D.	F	M	M	M	M	F
Analyseur dynamique	H	M	F	H	M	F
Analyseur de standard	F	H	M	M	M	M
Analyseur statistique	F	M	F	M	M	M
Analyseur de temps	F	M	F	H	F	F
Debugger	M	F	F	H	F	F
Driver de test	F	F	F	H	H	F
Éditeur	F	M	F	M	M	F
Éditeur de texte	F	H	F	F	H	H
Flowcharter	F	M	M	M	H	H
Librairie	F	F	F	F	F	H
Moniteur de logiciel	H	F	F	H	M	F
Processeur de langage	M	H	H	H	H	H
Simulateur	H	M	M	M	M	H
Standard	H	H	H	H	H	H
Table de décision	F	F	F	M	M	F
Test Bed	H	H	H	H	H	H

Légende : H-Très Efficace, M-Efficacité Moyenne, F-Faible Efficacité.

Tableau 6.3 : Efficacité des outils pour vérifier les caractéristiques de la qualité.

Le *tableau 6.3* montre que la standardisation et les tests beds permettent de contrôler très bien la qualité. Les tests sont la fonction de SQA la plus supportée par les outils. La standardisation et l'éditeur de texte sont utilisés pour toutes les fonctions du SQA.

	Fonctions de SQA.				
Outils	Planning de la qualité	Tests	Standards de design	Standards de documentation	Reviews et audits
Analyseur de B.D.		×			
Analyseur dynamique		×			
Analyseur de standard		×	×	×	
Analyseur statistique		×	×	×	
Analyseur de temps		×			
Debugger		×			
Driver de test		×			
Éditeur		×	×		
Éditeur de texte	×	×	×	×	×
Flowcharter		×			
Librairie		×			
Processeur de langage		×	×		
Moniteur de logiciel		×			
Simulateur		×	×		
Standard	×	×	×	×	×
Test Bed		×			
Table de décision		×			×

Tableau 6.4: Comment les outils supportent les fonctions de SQA.

Face à tous ces outils et techniques, il faut choisir les plus efficaces tout en se préoccupant de leur prix. Certains sont indispensables à un programme de SQA: pour les techniques, ce sont l'auditing, les inspections de design, les reviews, la standardisation, le walk-throught. Pour les outils, l'analyseur dynamique, les processeurs de langage, les standards, l'analyseur de standards et le bed test apparaissent indispensables.
Dans les années à venir, les outils devraient passer d'un rôle passif à un rôle actif, comme alerter le manager d'un événement et l'aider à prendre des décisions. Pour cela, les outils devront pouvoir collecter l'information adéquate, ce qu'ils ne font pas actuellement. Il faudrait alors qu'ils soient intègres afin que les outputs de certains outils puissent être les inputs d'autres. Les données pourront alors être partagées.

CHAPITRE 7:

LA SOFTWARE QUALITY ASSURANCE (SQA).

On va s'intéresser dans ce chapitre au niveau organisationnel du processus de développement de logiciel et voir comment améliorer la qualité d'un logiciel grâce à la SQA (Software Quality Assurance). Il a été observé que lors du développement de logiciel sans SQA, il y avait des problèmes au niveau de la qualité. Malheureusement, bien souvent l'organisation de SQA n'existait pas.

Pour le développement de hardware, une organisation de Quality Management a été nécessaire afin de délivrer un bon produit alors que les logiciels ont été historiquement développés de manière informelle. Dans les premières années après l'apparition des projets de logiciel, ce sont des ingénieurs, des mathématiciens et des personnes très créatives de milieux différents qui ont travaillé dans ce domaine. Le revers de la médaille est que les logiciels ont été développés dans le flou artistique, avec un manque de discipline, ce qui n'a pas été le cas pour le hardware. Même si par la suite les écoles ont enseigné des méthodes pour le développement de logiciel (qui ne sont d'ailleurs parfois pas appliquées), il n'en reste pas moins qu'il reste des traces des pratiques des premiers développeurs de logiciel.

Ce n'est que vers les années 74 que les gouvernements et les industries ont reconnu un besoin de SQA. On a fini par s'apercevoir qu'un logiciel était tellement important pour les performances d'un système qu'il fallait accorder beaucoup d'attention à sa qualité.

7.1 Présentation de la "Software Quality Assurance".

L'organisation se compose de différents départements dont le PMO (Project Management Office), le laboratoire des programmeurs et le département de SQA.

La SQA peut être défini ainsi:

"La SQA est un programme d'activités planifiées pour déterminer, aboutir et maintenir la qualité requise pour un logiciel, au niveau de toute l'organisation. Le but de la SQA est d'améliorer la productivité, la qualité du produit et du processus de développement".

Ce sont les managers qui doivent planifier pour arriver à un logiciel de qualité, revoir l'objectif constamment et être assez flexibles pour modifier le plan original. La SQA regroupe toutes les fonctions de niveau engineering pour assurer la qualité d'un logiciel. Sans une organisation de SQA, il ne sera pas défini clairement qui doit s'assurer de la qualité du logiciel.

Pour établir un programme d'assurance de la qualité, l'organisation de SQA doit être indépendante et séparée de l'organisation de développement du logiciel, les objectifs de la SQA n'étant pas les mêmes que ceux des développeurs (qui veulent parfois prendre une certaine liberté). La SQA doit alors soutenir plus les besoins du client que les besoins définis par les développeurs du projet (il peut alors y avoir conflit avec les programmeurs). C'est pourquoi la décision d'avoir une organisation de SQA doit venir du top-management, qui doit donner des instructions et des directions au personnel du département de SQA.

Il faut tout de même nuancer cette approche car pour de petits environnements, avec peu de programmeurs et sur une période courte, il est possible de délivrer un logiciel de qualité sans avoir une SQA séparée qui ici serait contre indiquée, coûtant trop cher.

7.2 Un modèle quantitatif de SQA.

Le département de SQA ne suit pas une méthodologie standard et chaque organisation a ses propres spécificités. Mais un département de SQA doit toujours se poser les 4 questions suivantes (appelées les 4 W' du SQA):

- What: Que doit-on assurer? Il faut identifier les caractéristiques de qualité, et voir les relations entre ces caractéristiques[22] . La SQA doit établir comment mesurer ces caractéristiques et définir des valeurs attendues de métriques, en relation étroite avec le client.

- When: Quand doit-on mesurer? Il faut identifier les étapes où les caractéristiques de qualité sont contrôlées pour s'assurer que les besoins sont satisfaits (ou prendre des actions correctives si les besoins ne peuvent être satisfaits).

[22] Voir le chapitre 1.

- Which: Quelles méthodes et outils utilisés pour rassembler l'information pour supporter les activités de SQA[23]? Les outils doivent permettre de réduire la charge de travail indirect des programmeurs. Un des problèmes important en SQA est de garder trace des problèmes. Si ce travail est laissé aux programmeurs et qu'on leur impose de faire des rapports, ils vont trouver que c'est un fardeau. D'ailleurs, bien souvent, on n'obtient pas beaucoup d'informations des programmeurs sur l'évolution d'un projet. L'idée est de les libérer des tâches non directement liée à la programmation, en créant des procédures pour donner aux programmeurs et au management des informations sur le développement. On peut alors créer une B.D. où seront stockées les améliorations futures et les prédictions concernant la programmation. Il est également judicieux de placer les documents générés par les programmeurs sous contrôle de configuration et ceci à la fin de chaque phase. On ne peut alors effectuer des modifications que sur permission et cela évite que les spécifications ne soient plus respectées.

- Who: Qui doit faire l'assurance, quelle sorte de profession, avec quelle qualification et avec quelles relations avec le personnel du projet?

Le modèle quantitatif de SQA proposé comporte les 3 phases suivantes:

1. Définir les besoins de qualité en termes quantitatifs et objectifs: spécifier que le temps de réponse doit être acceptable n'est pas un besoin mesurable. Il faudra alors mieux définir un temps maximal de réponse.

2. Planifier le contrôle de la qualité: c'est à dire faire un plan des actions pour s'assurer que les besoins sont remplis, contrôler la propre exécution de ces actions et évaluer les résultats. Le plan doit définir comment quantifier et contrôler les caractéristiques de qualité, comment collecter les données et quelles méthodes et outils utilisées. Le plan doit être établit par les utilisateurs et l'équipe de développement. Beaucoup de programmeurs sont opposés à ce plan, à cause de la petite taille du projet ou pour d'autres raisons comme la perte de contrôle sur le travail. Le plan peut être mis à jour durant le projet.

3. Exécuter le contrôle de la qualité: cela consiste d'abord à effectuer des mesures à l'aide de méthodes et techniques définit par le plan puis à comparer les mesures avec les valeurs attendues.

[23] au chapitre 6, des méthodes et outils ont été présentés.

7.3 Les Activités de la SQA.

Voyons maintenant quelles sont les activités que devrait avoir tout département de SQA:

- Le Planning: cela consiste à préparer un plan qui interprète les demandes de qualité et assigne des tâches, organise les responsabilités.

- Le Développement de Procédures et de Pratiques: cela consiste à préparer des standards[24] pour toutes les phases de développement du logiciel. Le point crucial est de faire attention très tôt à la qualité dans le développement d'un logiciel car on fait des économies en découvrant les erreurs tôt. On a constaté qu'il y avait un manque de standards et de terminologies en SQA, ce qui peut entraîner des confusions. Par exemple, tous les programmeurs n'ont pas la même conception de la définition d'un "milestone": pour certains c'est une date précise, pour d'autres c'est un événement (comme la fin d'une phase). Mais le IEEE Computer a établit un projet: "standard for SQA plan".

- Le Développement d'Aide et d'Assurance de Qualité: cela consiste à développer des procédures pour vérifier la conformité des fonctions du logiciel et de la qualité requise.

- Les Audits: ils concernent la revue des standards de développement.

- La Surveillance des Tests: cela consiste à faire des rapports sur les problèmes du logiciel, à analyser les causes d'erreurs et à s'assurer de leur correction.

- Le Maintien des Enregistrements: cela consiste à conserver des rapports sur les problèmes du logiciel, son design, sur les tests.

- Le Contrôle du Matériel Physique: cela consiste à inspecter les disques, les cartes pour s'assurer que leur contenu n'est pas détruit ou altérer par l'environnement ou une mauvaise manipulation.

[24] voir chapitre 6.

7.4. Évaluation de la Qualité d'une SQA.

Barney M.Knight a défini 3 attributs importants pour évaluer la qualité d'une organisation de SQA:

- Avoir une SQA qui s'intègre bien avec toute l'organisation: c'est l'attribut le plus important. Il faut développer des interfaces entre la SQA et tous les autres départements afin que l'information circule bien et qu'il n'y ait pas de redondances au niveau des tâches. Il sera utile d'avoir un diagramme avec la SQA au centre et les départements autour afin de montrer les missions de chacun. Le client est une interface importante pour la SQA. Il faudra partager ouvertement les informations concernant la qualité tout au long du développement du logiciel.

- Avoir une crédibilité technique: comme il faut beaucoup d'expertise et de jugement technique pour collecter et analyser les bonnes données, la SQA aura besoin d'un personnel technique compétent. Il faudra alors insérer des programmeurs dans la SQA et enseigner aux ingénieurs de qualité les principes de programmation.

- Avoir des coûts justifiés: le manager de la SQA doit s'interroger sur les coûts qu'engendrent son organisation tout au long du développement du logiciel et se demander s'ils sont justifiés. Il est alors intéressant de voir la répartition des coûts selon les activités de SQA et de les comparer aux coûts de développement. En général, les coûts sont répartis ainsi: 16% des coûts en meeting et coordination, 15% en contrôle et vérification de tests, 13% en assurance de configuration de logiciel (contrôle des librairies, inspection de la construction du système), 12% en présentation et écriture. Pour un petit projet, les coûts de SQA représente 12% des coûts de développement, 4% pour un grand projet.

Il est possible d'estimer les coûts de la SQA sans avoir à rassembler beaucoup de données. Il existe 2 catégories de coûts: les coûts de départ et non récurrents (génération d'un plan initial, développement d'outils, de procédures) et des coûts récurrents (reviews, inspections, audits, etc).

CONCLUSION.

Le but du "software engineering" est de produire des logiciels de qualité à moindre coûts. Ce but ne pourra être atteint sans l'utilisation de mesures objectives de la qualité: sans les métriques, une organisation pourra difficilement voir si elle a atteint ses objectifs en matière de qualité.

Les métriques et les modèles sont utilisés:

- Pour décrire des buts quantitativement.
- Pour décrire quantitativement l'état courant de la qualité d'un logiciel.
- Pour prédire le coût d'un système, sa "reliability" et sa date de livraison.

L'histoire des métriques a fortement été influencée par l'évolution du développement des logiciels:

- Les années 1970 marquent l'émergence de la programmation structurée. Les métriques basées sur le flux de contrôle du programme apparaissent, la plus populaire étant la complexité cyclomatique de Mac Cabe [25].
- Les années 1980 mettent au premier plan les phases de spécifications et de design. La programmation apparaît moins importante. Les métriques de design (la plus citée étant celle de Henry et Kafura) sont développées à cette époque.

Actuellement, il n'y a pas de métriques et modèles fiables et précis. La plupart sont basés sur des données d'un environnement particulier et ne donnent pas de mesures exactes pour toutes les organisations. Beaucoup de recherches et d'études sont encore à faire pour valider et redéfinir les modèles et métriques existants et en définir de nouveaux.

[25] Voir chapitre 2.

Le problème majeur dans le développement des métriques est le manque de données précises concernant l'industrie du logiciel. Le DOD[26] reconnaît qu'un effort est nécessaire pour collecter un ensemble de données communes à partir d'un grand nombre de projets. Ainsi pourra-t-on développer des métriques valides pour toutes les organisations.

Certains ingénieurs de logiciel affirme que des attributs comme la qualité ou l'utilité ne sont pas mesurables, ne pouvant être définis exactement. Nul ne peut affirmer que ces attributs ne seront pas un jour mesurés objectivement. En effet, en Sciences Physiques, en médecine, il est possible de mesurer aujourd'hui des attributs comme la température alors qu'on pensait auparavant que c'était impossible. Mais il faudra pour cela disposer de définitions précises des attributs et des métriques (le chapitre 2 a mis en évidence que des métriques comme le nombre de lignes de code n'avaient pas la même définition dans toutes les organisations).

Les métriques suscitent depuis récemment un intérêt particulier dans l'industrie même si elles sont encore peu utilisées. Les métriques les plus populaires sont les mesures de tests et les modèles d'estimation des coûts comme COCOMO.

En général, les managers sont réticents à utiliser les métriques pour les raisons suivantes:

- Les métriques et les modèles ne sont pas communément acceptés.
- Les managers ne sont pas prêt à dépenser de l'argent dans un programme de mesure de la qualité sans être convaincus de son utilité.
- Peu d'articles sur les métriques sont destinés aux industriels. Une meilleure diffusion des métriques et des modèles est nécessaire au niveau des managers.
- Peu d'outils calculant des métriques sont diffusés. Seulement 2 outils étaient commercialisés en 1990: Logiscope et Qualigraph (qui calcule les mesures d'Halstead, la complexité cyclomatique et des mesures du flux de données et de contrôle).

Pour mettre en place un programme de SQA à faible coût, il est alors nécessaire de développer des outils comme Datrix[27] qui collectent automatiquement des données pour fournir des métriques. Des programmes d'enseignement sur l'utilisation des métriques sont également attendus.

[26] Department Of Defense.
[27] Voir le chapitre5.

Pour améliorer qualité et productivité, de nouveaux paradigmes peuvent être développés. Balzer, Cheathon et Green [BALZ83] suggère d'exécuter la maintenance (qui coûte plus de 60% des coûts de développement) à partir des spécifications plutôt qu'à partir d'un code source optimisé et artificiellement complexe. Pour cela, l'implémentation doit être rapide, peu coûteuse et accompagnée d'un support automatique (comme un outil qui génère du code à partir de spécifications formellement définies). Cette approche intéressante est dans ces débuts.

BIBLIOGRAPHIE.

1. Basili V.R., Rombach H.D., « Implementing quantitative S.Q.A: a practical model », IEEE software, septembre 1987.

2. Boehm B.W., Brown J.R., Lipow M, « Quantitative evaluation of software quality », 2ème International Conference of Software Engineer.

3. Cavano J.P., Lamonica F.S., « Quality assurance in futur development environments », IEEE software, septembre 1987.

4. Collofello J.S., Buck J.J., « Software quality assurance for maintenance », IEEE software, septembre 1987.

5. Conte S.D., Shen V.Y., Dunsmore H.E., « Software engineering metrics and models », Benjamin Cummin Publishing inc., 1986.

6. Cooper J.D. et Fisher M.J. éditeurs: « Software quality management », Petrocelli, 1979.

7. Fenton N.E., « Software Metrics, a rigorous approach », Chapman Hall, 1991.

8. Grady R.B., « Measuring and managing software maintenance », IEEE software, septembre 1987.

9. Kishadfa K., Teramoto M., Torri K., Urano Y., « Quality assurance technologie in Japan », IEEE software, septembre 1987.

10. Mills H.D., Duer M., Linger R.C., « Cleanroom software engineering », IEEE software, septembre 1987.

11. Poston R.M., Bruen M.W., « Counting down to zero software failures », IEEE software, septembre 1987.

ANNEXE:

LA CRÉATION D'UNE B.D. POUR GÉRER LES REQUÊTES DE MODIFICATION.

RAPPORT DE DOCUMENTATION.

INTRODUCTION

Le logiciel documenté ici peut permettre d'améliorer la qualité. Il constitue un outil de gestion des requêtes de modification.

Actuellement, au laboratoire de Génie Logiciel de l'École Polytechnique de Montréal, il n'y a pas de système automatisé qui permette d'enregistrer aussi bien les modifications à effectuer sur les logiciels que les erreurs à corriger. Le but du projet est alors de créer une base de données sous Access (Microsoft) permettant d'automatiser la gestion des modifications. Il sera alors facile de produire des rapports ou d'interroger la base de données pour connaître toutes les modifications à effectuer sur un logiciel.

Le chapitre 1 présentera comment sont stockées manuellement les informations sur les modifications à effectuer.
Puis, Access permettant de créer des bases de données de type relationnel, on verra comment les tables on été implémentées et quels sont les liens entre elles.
On décrira ensuite comment entrer des données à tous les niveaux du processus de modification.

Comme on cherche également à produire des rapports sur les produits logiciels pour lesquels une requête de modification a été introduite, on présentera quelles informations il est possible d'obtenir.

CHAPITRE I:

DESCRIPTION DU SYSTÈME MANUEL.

Quand on découvre une erreur dans un logiciel ou encore lorsqu'on voudrait le modifier pour l'améliorer, on introduit une *requête de modification* en remplissant une fiche. On y spécifie:

- Le nom du produit logiciel (Schémacode par exemple).
- La version du logiciel.
- L'environnement du logiciel.
- Le langage à utiliser pour modifier le logiciel ou le langage à analyser pour corriger l'erreur.
- La fonctionnalité affectée.
- Le type de problème (un parmi les suivants: produit arrêté, résultats erronés, messages d'erreurs, erreurs de documentation, modification au produit, ajout au produit).
- Une description informelle du problème.
- Le nom de l'usager.
- La date.

Une analyse sera ensuite effectuée pour rechercher la cause de l'erreur ou voir comment procéder à la modification. C'est alors qu'on remplit une fiche *d'analyse de la requête.*
Cette fiche reprend :

- Le nom du vérificateur.
- L'identification de l'erreur ou de la modification.
- La date.
- Le type de requête : correction d'erreur, modification de fonctionnalité ou ajout de fonctionnalité.
- La phase du développement où l'erreur ou la modification a été détectée : (analyse, design, implémentation, etc).
- Le matériel utilisé.
- Le logiciel utilisé.
- Une description.
- Les causes probables.
- L'impact sur la réalisation des tests (empêche ou non de poursuivre).
- L'impact sur l'échéance du projet (retard majeur, mineur...).
- La priorité.

Ce n'est qu'après cela que l'on pourra effectuer la modification. Il faut noter qu'il est possible qu'on ait un bug connu mais jamais corrigé.

Si le code est modifié, il faudra remplir un *rapport de modification.* On y spécifiera:

- Le nom de la personne qui a effectué la modification.
- L'identification de l'erreur ou de la modification.
- La date.
- La cause (code manquant, incorrect...).
- Le type de faute (algorithmique, modification...).
- La condition d'occurrence (situation rare...).
- La sévérité (changement d'architecture majeur, mineur...).
- L'effort requis pour la modification en heures-personne.
- La phase où l'erreur a été commise.
- Les routines impliquées.
- La description de la modification.
- Des commentaires.

Pour résumer, on effectuera d'abord une requête de modification qui est toujours suivie d'une analyse. Selon le cas, on aura un rapport de modification ou non.

Grâce à une automatisation, il sera possible de produire des rapports: par exemple indiquer pour un produit les bugs corrigés et les bugs connus.

CHAPITRE 2:

La Création de la B.D.

Dans ce chapitre, toutes les tables de la base de données seront présentées.

La table REQUETE contient des informations sur les requêtes enregistrées que l'on n'a pas rejetée, c'est à dire qui sont traitées ou qui devront l'être.

Table: Requete de Modification

Field Name	Data Type	Description
Id_Req	Date/Time	Identifie une requete de modification par la date et l'heure.
Nom_Produit	Text	C'est le nom de la partie de logiciel à modifier.
Version	Number	C'est la version du logiciel.
Environnement	Text	C'est l'environnement du logiciel.
Langage	Text	C'est le langage utilisé dans la partie du logiciel à modifier.
Fonctionnalité	Text	C'est la fonctionnalité affectée.
Problème	Number	Ici, on stocke un nombre indiquant la nature du pb.
Description	Memo	On peut entrer une description non formelle.
Nom_usager	Text	Nom de la personne qui introduit la demande.
Priorité	Number	De 1 (+ urgent) à 5.

Table 2.:1 Requête de modification.

Id_req est un identifiant pour cette table. C'est en fait la date + l'heure système (on suppose que 2 requêtes n'arrivent pas à la même seconde).

Pour effectuer des recherches dans la base de données, on associe 1 ou plusieurs mots-clés à une requête. Du fait de cette relation many-to-many, on a du construire une autre table: MOT_CLE Ici, Id_req n'est pas identifiant puisque qu'une même requête peut avoir plusieurs mots-clés. On pourra par contre retrouver les mots-clés associés à une requête, puisque Id_req est une clé de cette table.

Table: Mots-Clés d'une Requete

Field Name	Data Type	Description
id_req	Date/Time	identifiant d'une requete.
mot_clé	Text	mot-clé associé à une requete.

table 2.2: Mots-clés d'une requête.

Pour introduire les données relatives à la fiche d'analyse de la requête, on dispose de la table ANALYSE:

Microsoft Access - [Table: Analyse d'une Requete]

File Edit View Window Help

New:

Field Name	Data Type	Description
Id_req	Date/Time	Identifiant de la requete.
Nom_verificateur	Text	Nom de la personne qui a effectuée l'analyse.
Identification_erreur	Text	On intoduit le type de l'erreur ou de la modification.
Date	Date/Time	Date à laquelle on introduit les données.
Type_requete	Number	Correction d'erreur, modification ou ajout de fonctionnalité.
Phase_origine	Number	Phase du développement où l'erreur a été détectée.
Matériel	Text	Equipement utilisé.
Logiciel	Text	Equipement utilisé.
Description	Memo	Non formel.
Causes_problables	Memo	Non formel.
Réalisation_test	Number	Empeche ou non de porsuivre.
Echéancier	Number	Retard majeur, mineur, imprévisible ou aucun retard.
Priorité	Number	Priorité pour modifier (de 1 à 4).

Table 2.3: Analyse d'une requête.

Id_req est un identifiant de cette table. A partir de l'identifiant d'une requête, on peut ainsi trouver son analyse, si elle existe.

Si par la suite on effectue une modification dans le logiciel, on stockera les informations relatives à la fiche «*rapport de modification*» dans la table suivante :

Microsoft Access - [Table: Rapport de Modification]

File Edit View Window Help

New:

Field Name	Data Type	Description
Id_Req	Date/Time	Identifiant de la requete.
Nom	Text	Nom de la personne qui a effectuée la modification.
Ident_Erreur	Text	Identification de l'erreur.
Date	Date/Time	Date de l'introduction du rapport.
Cause	Number	Code manquant, incorrect, mort ...
Type_Faute	Number	Algorithmique,documentation,fonctionnelle...
Condition_d'Occurence	Text	Situatuins rares, timing ou autre...
Sévérité	Number	Changement dans l'architecture majeur, mineur...
Efforts_Requis	Number	Efforts requis pour modifier en heures-personne.
Phase_Erreur	Number	Phase où l'erreur a été commise.
Routines_Impliquées	Memo	Non formel.
Description_Modification	Memo	Non formel.
Commentaires	Memo	non formel.

Table 2.4: Rapport de modification.

Id_req est encore un identifiant. Comme c'est une clé, on pourra par exemple, à partir de la table requête de modification voir si un bug connu a été corrigé.
Il se peut que l'on trouve que 2 requêtes entrées par différentes personnes qui traitent du même problème. Afin de ne pas perdre d'information, on ne veut pas effacer une des 2 requêtes. On va alors les grouper, c'est à dire que si l'une est traitée, l'autre l'est aussi. Les 2 requêtes ont la même analyse et le même rapport. On crée alors une table: GROUPE_REQUETE:

Req_pere	Date/Time	l'identifiant de la requête "chef" du groupe. Cet identifiant caractérise l'analyse et le rapport des requêtes du groupe.
Req_fils	Date/Time	une des requêtes du groupe identifié par Req_pere.

Table 2.5: Groupe requête.

Lors de l'enregistrement d'un groupe, il faut choisit une requête comme reqpere. Elle est ainsi désignée comme chef du groupe. On choisit aussi une requête comme reqfils. Lors de l'enregistrement d'un rapport ou d'une analyse, un record est crée avec comme identifiant celui du reqpere. Si on cherche le rapport ou l'analyse d'un req_fils, on doit regarder dans les tables analyse et rapport le record avec comme identifiant celui du req_pere. Si on veut mettre plus de 2 requêtes dans un groupe, on garde toujours le même req_pere.
Par exemple, si on veut mettre les requêtes 1,2,3 dans le même groupe, on choisit une requête comme reqpere. Disons la requête 1: on aura alors les records suivants dans la table GROUPE_REQUETE:

Req_pere	Req_fils
1	2
1	3

Un req_fils ne peut pas être req_pere. on ne pourra pas alors avoir:

Req_pere	Req_fils
1	2
2	3

2 requêtes ne peuvent être groupées si elles ont déjà été analysées toutes les 2.
Si on cherche à grouper 2 requêtes dont une a déjà été analysée, on devra enregistrer la requête analysée comme req_pere. La table a comme identifiant Req_fils, une requête fils ne pouvant faire partie que d'un seul groupe.

Si on considère qu'une requête enregistrée ne devra pas être traitée, une modification n'étant pas nécessaire, elle sera rejetée. On utilisera la table REQUETE_REJETEE. Elle contient les même champs que la table REQUETE avec en plus le champs Raison.

Id_req	Date/Time	Identifie une requete de modification par la date et l'heure.
Nom_produit	Text	C'est le nom de la partie de logiciel à modifier.
Version	Number	C'est la version du logiciel.
Environnement	Text	C'est l'environnement du logiciel.
Langage	Text	C'est le langage utilisé dans la partie du logiciel à modifier.
Fonctionnalité	Text	C'est la fonctionnalité affectée.
Problème	Text	Ici, on stocke la nature du pb.
Description	Memo	On peut entrer une description non formelle.
Nom_usager	Text	Nom de la personne qui introduit la demande.
Priorité	Number	De 1 (+ urgent) à 5.
Date	Date/Time	Date de la requête (ce n'est pas obligatoirement la date d'introduction de la requête.
Raison	Memo	La raison du rejet de la requete.

Table 2.6: Requête rejetée.

Il y a également 5 tables temporaires qui servent à garder de l'information lorsque l'on demande un rapport général, c'est à dire lorsque l'on fait une recherche de requêtes suivant différents critères.

Les tables LISTE ET LISTE_TEMPORAIRE ont exactement les mêmes champs que la table REQUETE_REJETEE. Elles sont utiles pour trouver les requêtes ayant les critères introduits (voir plus loin).

La table CHOIX_ID permet de mémoriser un ensemble de requêtes. On l'utilise entre autre pour proposer les requêtes pouvant être req_pere, req_fils.

Id_rec	Date/Time	cette table contient des identifiants.

Table 2.7: Choix ID.

Les 2 tables MC_RECHERCHE et MC_NON_RECHERCHE servent respectivement à garder en mémoire les mots-clé recherchés et non recherchés, lorsque l'on veut obtenir les requêtes répondant à certains critères.

Mot_Clé	Text	cette table contien les mots_clé non recherché quand on demande un rapport general.

Table 2.8: Mots-clé recherchés.

CHAPITRE 3:

LA SAISIE DES DONNÉES.

Le menu principal propose les différents choix suivants :
En ce qui concerne l'entrée de données :
- Enregistrer des Requêtes de Modifications, des Analyses, des Rapports ...
- Grouper des Requêtes.
- Rejeter une Requête.

En ce qui concerne les rapports:
- Voir la liste des Requêtes de Modification Non-Réalisées.
- Voir la liste des Requêtes de Modification pour une Version donnée et un Produit donné.
- Editer un Rapport général.

3.1. Enregistrer des Requêtes de Modifications, des Analyses, des Rapports.

3.1.1 Ajout, Effacement et Modification de Requêtes.

Commençons par le début et voyons ce qui se passe si on choisit d'enregistrer une Requête de Modification. En poussant le bouton de commande adéquate à partir du menu général , la forme ADD REQUETE, reliée à la table REQUETE (qui contient toutes les Requêtes non-rejetées) s'ouvre en mode Edit, c'est à dire que l'on peut voir tous les records de la table REQUETE, les modifier et en insérer de nouveau.
Par défaut, la forme en s'ouvrant se positionne sur le premier record de la table REQUETE.

Lors de la saisie d'un champs à l'aide d'une combo box, on peut voir les différentes valeurs non nulles des records de la table pour ce champs. On utilise alors une clause SQL comme source pour la combo box. Par exemple, la Row source de la combo box associée au nom de produit est : Select distinct Nom_produit from requete where Nom_produit is not null.

Pour d'autres champs comme la priorité ou le type de problème, on utilise une list box ou il y a toujours une valeur par défaut.

Si on modifie la valeur des champs et que l'on passe à un autre record (à l'aide des flèches permettant de naviguer d'un record à l'autre), la modification sera enregistrée. On peut utiliser aussi le bouton commande **Save**, qui va appeler le macro ADD REQUETE qui va effectuer l'action DoMenuItem: Save Record.

Pour enregistrer une nouvelle Requête de modification, il suffit de presser le bouton de commande **New**, qui permettra de se positionner sur un nouveau record à la fin de la table REQUETE. (Mais comme cette table est indexée par id_req, elle sera par la suite triée selon ce champs).
Par défaut, la valeur de Id_req est égale à la date et l'heure du système. Mais il est possible de changer ces valeurs. Seulement, lors de l'enregistrement du record, on vérifiera si Id_req n'est pas nul et s'il n'est pas en double.

Pour effacer le record courant, il suffit de presser le bouton **Del**. Le macro Del Requete est alors appelé. On vérifie alors si le record courant a déjà été analysé. On utilise la fonction Dcount qui retourne un nombre.
DCount("[Id_req]","ANALYSE","[Id_req]=forms![ADD REQUETE]![Id_req]")<>0
Cette expression est vraie si le nombre de record dans la table Analyse ayant le même identifiant que celui du record courant est strictement supérieur à zéro. Dans ce cas, le macro est arrêté après avoir indiquer un message d'erreur.

De même, pour effacer une requête qui fait partie d'un groupe, un message indiquant qu'il faut d'abord retirer la requête de son groupe est affiché. La forme ADD GROUPE REQUETE s'ouvre alors en mode dialog (la fenêtre garde le contrôle jusqu'à ce qu'elle soit fermée).
Si la requête ne fait pas partie d'un groupe, si elle n'a pas été analysée et si elle a été enregistrée (il se peut qu'on appelle Del alors qu'on est en train d'enregistrer une nouvelle requête et qu'on ne l'a pas encore enregistré), le macro del Requete s'arrête après avoir appelé la forme Confirm del qui va permettre de confirmer l'effacement du record.
On pose alors la question: On efface le record? On a le choix entre Cancel
(On ferme la fenêtre en poussant ce bouton commande) et Ok qui appelle le macro Del Requête 2.
2 clauses SQL sont alors lancées:

- Effacer le record concerné dans la table REQUETE :(Delete * from REQUETE where Id_Req= Forms![ADD REQUETE]![Id_Req];)

- Effacer les records dans la table MOT_CLE.
  delete * from Mots_cle where id_req=fprms![add requete]![Id_req];

On peut s'aider du find en pressant le bouton commande **FIND** qui appelle le macro FIND qui se contente de faire un DoMenuItem find (c'est la même chose que d'appeler Find à partir du menu).

3.1.2 Enregistrer des mots-clé à une requête.

Ayant enregistré notre requête ou ayant modifié une requête existante, on peut alors vouloir lui associer un ou plusieurs mot-clé. Il suffit pour cela de presser le bouton action **Mot_Clé** qui appelle le macro APPEL MC. Le système vérifie alors (par une fonction Dcount) que la requête courante a effectivement été enregistrée et si c'est le cas, le système ouvre la forme ADD MOT CLE en mode dialog. Cette forme est associée à la table MOT_CLE.

Une list box affiche alors tous les mots-clé déjà enregistrés (par une clause SQL) et une combo box indique les mots-clés de la requête courante dans la forme ADD REQUETE. Une boite de dialogue indique par défaut la requête courante mais on peut sélectionner un autre identifiant. Bien sur il est impossible d'entrer de nouvel identifiant.

Il est possible d'enregistrer un nouveau mot-clé en appelant le macro ADD MC. Le système vérifie d'abord que le mot-clé n'a pas déjà été enregistré puis que le mot-clé n'a pas une valeur nulle. Le record est alors sauvé par un DoMenuItem: Data Entry et le système met à jour les combo et list box par des Requery.

Pour effacer un mot-clé, le système doit appeler le macro DEL MC qui efface le record par une clause SQL, recalcule les champs et se positionne sur un nouveau record.

Ayant enregistré des mots-clés à une requête, on peut vouloir la rejeter. Il faut alors appeler alors le macro APPEL REJET2 toujours en appuyant sur un bouton commande. Ce macro, après avoir vérifié que la requête n'a pas été analysée et qu'elle ne fait pas partie d'un groupe, copie le record de la requête courante de REQUETE dans la table intermédiaire LISTE. En effet c'est cette table qui est associée à la forme REJET REQUETE. Nous verrons plus loin pourquoi. On ouvre la forme REJET REQUETE (read only) en se positionnant sur le record de la table REQUETE qui a le même identifiant que celui dans la forme ADD MOT CLE. Il faut noter que l'on peut aussi rejeter une requête à partir du menu principal. Seul l'appel de la forme REJET REQUETE diffère. Pour plus d'information sur le rejet d'une requête, voir plus loin...

Lors d'un enregistrement de mots-clé, on peut vouloir grouper des requêtes, si elle ont les mêmes mot-clés par exemple. On lance alors le macro APPEL GROUPE REQUETE qui va ouvrir la forme ADD GROUPE REQUETE en mode dialog. Une fois de plus, on peut aussi appeler cette forme depuis le menu principal, ce qui aura le même effet.

Quand la gestion des mots-clé est terminée, on peut alors fermer la forme ADD MOT CLE par un bouton commande, ce qui appellera le macro CLOSE.
Ce macro ferme la forme active sans enregistrer le record courant s'il ne l'avait pas été. Ceci peut éviter des sauvegardes non désirées. Par contre, si on ferme la forme en utilisant le menu de la boite de dialogue en haut à gauche, le record courant sera sauvé s'il ne l'était pas.

3.1.3 Consulter oueEnregistrer l'analyse d'une requête de modification.

Si maintenant, on veut voir ou enregistrer une Analyse de la Requête de modification courante, on peut presser le bouton commande **Analyse** qui va lancer le macro APPEL ADD ANALYSE. (Rappelons à cet effet que toutes les requêtes d'un groupe ont la même analyse)

Ce macro va d'abord vérifier que la requête courante a effectivement été enregistrée. Puis le macro va ouvrir la forme ADD ANALYSE avec des champs vides si la requête courante n'a pas été analysée. Pour cela, il faut compter (avec Dcount) le nombre d'éléments du query RECHERCHE ANALYSE.

Si la requête courant a été analysée, le query va sélectionner le record d'ANALYSE qui correspond, sinon la requête n'a pas été analysée et le query est vide. Plus précisément, le query sélectionne le record dans la table ANALYSE qui a un identifiant identique au record courant dans la forme ADD REQUETE ou sélectionne le record dans ANALYSE dont l'identifiant correspond à une requête père qui a une requête fils dont l'identifiant est le courant. (Il ne faut pas oublier que quand on enregistre un groupe de requêtes, on a un identifiant de requête qui est appelé req-père qui caractérise le groupe et les autres identifiants du même groupe sont des reqfils. Toutes les requêtes de ce groupe auront la même analyse et le même rapport de modification).

Si la requête courante n'a pas été analysée et ne fait pas partie d'un groupe en tant que req_fils, le système ouvre la forme ADD ANALYSE avec le même identifiant que celui de la forme ADD REQUETE.

Par contre, si la requête courante n'a pas été analysée et est une req_fils, le système ouvre la forme avec la req_pere comme identifiant.

ADD ANALYSE étant ouverte, on peut modifier le record courant ou alors enregistrer une nouvelle analyse (il y a le bouton commande **Save** qui appelle le macro ADD ANALYSE qui sauve le record et ferme la forme ADD ANALYSE). Pour éviter d'enregistrer 2 analyses pour une même requête, dès que l'on essaie de passer à un autre record, le macro GOTO BEGIN est lancé. Comme son nom l'indique, il revient au premier record de la forme ADD ANALYSE.

3.1.4 Consulter ou enregistrer le rapport d'une requête de modification.

C'est exactement la même chose que pour une analyse sauf que l'on vérifie que la requête a été effectivement analysée. Le macro lancé est APPEL ADD RAPPORT et la forme ouverte est ADD RAPPORT. Le query qui contient le record de la table RAPPORT correspondant à l'identifiant courant est RECHERCHE RAPPORT.

3.2. Grouper des Requêtes.

A partir du menu principal, il est possible de grouper des Requêtes en lançant le macro APPEL GROUPE REQUETE. La forme ADD GROUPE REQUETE apparaît. Elle est associé à la table GROUPE_REQUETE.

Pour s'aider à grouper les Requêtes, on peut voir les mots-clé de 2 requêtes en simultané. On dispose en effet de 2 combo box qui n'acceptent pas de nouveaux identifiants et qui proposent tous les identifiants des requêtes. A chaque combo box est associée une list box qui montrent les mots-clés de l'identifiant de la combo box associée.
Dès que la valeur d'une de ces combo box est changée, le système appelle le macro REQUERY MC qui va recalculer les list box.

Il est possible de consulter les groupes de requêtes existants. On dispose d'une list box qui affiche tous les différents req_pere existants dans la table GROUPE_REQUETE. Une autre list box affiche les valeurs des req_fils qui sont associées au req_pere sélectionné. Dès qu'une valeur de req_pere est modifiée, le système lance le macro REQUERY REQFILS qui recalcule les req_fils.

Rappelons que la table GROUPE_REQUETE contient 2 champs : req_pere qui est l'identifiant "père" d'un groupe et req_fils qui correspond à une autre requête du même groupe que req_pere.

Par exemple, pour grouper les requêtes 1, 2 et 3 en choisissant 1 comme req_pere du groupe, on aura 2 records:

req_pere	req_fils
1	2
1	3

Pour effacer une association de 2 requêtes, il suffit de sélectionner une valeur pour req_pere et une autre pour req_fils et d'enclencher Del. Le macro DEL GROUPE REQUETE est alors lancé et après avoir vérifier qu'aucun des 2 champs n'est nul, il va afficher une forme de confirmation (CONFIRM DEL GROUPE). Si on choisit OK, le système va lancer le macro DEL GROUPE REQUETE2 qui va effacer le record choisi dans GROUPE_REQUETE, fermer la forme de confirmation et recalculer les 2 lists box.

Si maintenant on veut grouper 2 requêtes de modification, on utilise les 2 combo box qui sont reliées à la table GROUPE_REQUETE.
Tout d'abord, on a la combo box qui permet d'enregistrer la valeur de Req_pere.
Cette combo box propose toutes les valeurs de la table CHOIX_ID et ne permet pas d'entrer de nouvelles valeurs. Dès qu'elle a le focus, le macro CHOIX REQPERE est lancé. Il doit remplir la table CHOIX_ID et pour cela commence par vider cette table, après avoir forcer un recalcul de la combo box (comme vous pouvez le constater, Access est plutôt fainéant et il faut toujours faire des requery après des modifications!).
2 queries sont ensuite lancés:

- CHOIX REQPERE qui se contente de copier dans la table CHOIX_ID tous les identifiants de la table REQUETE. (append query).
- CHOIX REQPERE2 qui efface de CHOIX_ID tous les numéros des requêtes fils (on doit faire les joints: CHOIX_ID.Id_req*REQUETE.Id_req et REQUETE.Id_req*GROUPE_REQUETE.Req_fils).

De plus, si la combo box qui propose le choix des Req_fils a une valeur, le système efface cette valeur de la table CHOIX_ID.

Justement, l'autre combo box va proposer des valeurs pour Req_fils. Elle tire ses valeurs de CHOIX_ID également et lance le macro CHOIX REQFILS dès qu'elle a le focus. Ce macro force un recalcul de la combo box et appelle le macro CHOIX REQPERE afin de mettre dans la table CHOIX_ID toutes les requêtes susceptibles d'être Req_pere. Mais il faut appeler le query CHOIX REQ FILS qui va effacer de cette table les identifiants qui sont Req_fils (on fait alors le joint : REQUETE.Id_req*GROUPE_REQUETE.Req_pere).
Le système ne doit pas également proposer comme req_fils les requêtes qui ont été analysées. Sinon, il serait possible de grouper 2 requêtes qui ont chacune 2 analyses différentes. Le système lance alors le delete query CHOIX REQFILS 2 qui efface de la table CHOIX_ID les identifiants de la table ANALYSE. (le joint est : CHOIX_ID.Id_req*ANALYSE.Id_req)

Si on a déjà choisi une valeur pour req_pere, le système la retire de la table CHOIX_ID avec une clause SQL comme d'habitude.
Il est alors impossible de grouper 2 requêtes qui ne devraient pas l'être.

Quand **Save** est enclenché, le macro ADD GROUPE REQUETE n'a qu'à vérifier que l'on a bien une valeur pour Req_pere et une valeur pour Req_fils et à enregistrer le record.

Quand on ferme la forme ADD GROUPE REQUETE, le système appelle un macro particulier: CLOSE DEL qui en plus de fermer la forme active nettoie aussi les tables intermédiaires (CHOIX_ID, LISTE, LISTE_TEMPORAIRE, MC_RECHERCHE, MC_NON_RECHERCHE).

3.3. Rejet d'une Requête.

Il se peut que finalement on décide de ne pas réaliser une requête de modification. Dans ce cas, il faudra la rejeter. Comme on l'a vu précédemment, on peut rejeter une requête à partir de la forme qui permet d'enregistrer les mots-clé.

Mais seules les requêtes qui n'ont pas été analysées et qui ne font pas partie d'un groupe de requêtes peuvent être rejetées .Quand la forme REJET REQUETE du menu général est appelée, le système lance le macro APPEL REJET qui va mettre dans la table intermédiaire LISTE toutes les requêtes non analysées. Pour cela, on vide la table liste par une clause SQL puis on lance le append query ALL REQUETE OK qui copie dans LISTE tous les records de REQUETE.

Puis le delete query REQUETE NON ANALYSEE1 efface de LISTE toutes les requêtes dont l'identifiant correspond à un record dans la table ANALYSE. On fait alors les joints: LISTE.Id_req * REQUETE.Id_req et REQUETE.Id_req * ANALYSE.Id_req.
On efface en fait toutes les requêtes qui ont un rapport "directement". mais il ne faut pas oublier qu'une requête fils a la même analyse que sa requête père. On doit également effacer ces requêtes. Pour cela le système appelle le delete query REQUETE NON ANALYSEE2 qui efface toutes les requêtes qui sont une req_fils et dont le req_pere a une analyse.
On fait les joints : (LISTE.Id_req*REQUETE.Id_req),
(REQUETE.Id_req*GROUPE_REQUETE.req_fis),
(GROUPE_REQUETE.Req_pere*ANALYSE.Id_req)

Le système ouvre ensuite la forme REJET REQUETE en mode read only, cette forme étant associée à la table LISTE. On ne peut alors choisir que des Requêtes non analysées. Ceci diffère de l'appel de cette forme à partir de l'enregistrement de mots-clé où le système ouvre la forme REJETE REQUETE seulement sur la requête courante si elle n'est ni analysée, ni membre d'un groupe.

La forme REJET REQUETE permet de voir tous les champs de la table LISTE qui, rappelons le à exactement les même champs que la table REQUETE. Si on veut faire une recherche, on peut activer le bouton commande FIND qui appelle le macro FIND. Ce macro, rappelons le, fait l'équivalent de l'item "Find..." dans le menu Edit.

Quand on est positionné sur le record de la requête à rejeter, il faut actionner le bouton commande Rejet, qui appelle le macro REJET REQUETE1. Le système vérifie alors que la requête courante ne fait pas partie d'un groupe. Si c'est le cas, le système ouvre la forme REJET REQUETE2 qui demande pour quelle raison on veut rejeter la requête. On peut néanmoins toujours faire un Cancel. Si on lance le macro REJET REQUETE en cliquant sur Ok, on va d'abord vérifier que l'on a effectivement rentré une raison. Si c'est le cas, le système va lancer des clauses SQL:

- insert into requete_rejetee select * from requete where [Id_req]=forms![rejet requete]![id_req]; { le système copie la requête à rejeter dans la table REQUETE_REJETEE qui contient les mêmes champs que la table REQUETE plus le champs :"Raison". Ce champs n'a pas encore de valeur }

- update requete_rejetee set Raison=forms![rejet requete2]![raison] where [Id_req]=forms![REJET REQUETE]![Id_req]; {le système met à jour le record nouvellement copié en enregistrant la valeur du champs raison}

- delete * from requete where [Id_req]=Forms![rejet requete]![id_req] {le système efface le record copie de la table REQUETE}
- delete * from liste where [Id_req]=Forms![rejet requete]![id_req] {et de la table LISTE.}

Quand on ferme la forme REJET REQUETE, le système appelle le macro CLOSE DEL qui va effacer les tables intermédiaires.

CHAPITRE 4:

LES RAPPORTS.

4.1 Voir la Liste des Requêtes réalisées pour une Version donnée.

En pressant ce bouton depuis le menu principal, le système appelle le macro APPEL DEMANDE VERSION. On vérifie alors qu'il y a bien des requêtes d'enregistrées puis on ouvre la forme DEMANDE VERSION en mode dialog. On dispose alors de 2 list box qui affichent les différents noms de produit et les différentes versions possibles. En cliquant sur OK dans cette forme, le système lance le macro APPEL REPORT PRO VERSION. On copie alors dans la table LISTE les requêtes qui ont un rapport et qui ont le nom de produit et la version indiquée.

Pour cela, après avoir vider la table LISTE, 2 queries sont lancés.
REQ PRODUIT VERSION1 qui copie dans LISTE les records de REQUETE dont l'identifiant correspond à un record de la table RAPPORT, avec le nom de produit et la version indiqués (joint: REQUETE.Id_req * RAPPORT.Id_req).

Il faut aussi copier les requêtes fils qui satisfont aux critères et dont le père a un rapport.
Le append query REQ PRODUIT VERSION2 fait les joints
REQUETE.Id_req*GROUPE_REQUETE.Req_fils
GROUPE_REQUETE.Req_pere*RAPPORT.Id_req.

Le système compte alors le nombre de requêtes dans LISTE et, s'il n'y en a pas, affiche un message d'information. Sinon, on ouvre le report REQ PRODUIT VERSION qui est associé à la table LISTE. La description est affichée et bien sur l'identifiant de la requête, en plus de la version et du nom de produit qui sont déjà connus.

4.2 Voir la liste des Requêtes de modification non réalisées.

Si on cherche maintenant les requêtes dont le traitement n'est pas encore terminé, il est possible d'obtenir la liste à partir du menu général.
Pour cela, il faut lancer le macro APPEL REPORT REQSANSRAPPORT. Encore une fois, la table LISTE est utilisé. Le système y copie d'abord toutes les requêtes de la table REQUETE puis lance 2 queries pour effacer les requêtes déjà réalisées.

REQUETE SANS RAPPORT1: on fait les joints: LISTE.Id_req*REQUETE.Id_req et

REQUETE.Id_req*RAPPORT.Id_req: on efface ainsi toutes les requêtes qui ont un rapport, sauf celles qui sont Req_fils.

C'est ce que fait:
REQUETE SANS RAPPORT2: on fait les joints: LISTE.Id_req*REQUETE.Id_req
REQUETE.Id_req*GROUPE_REQUETE.Req_fils
GROUPE_REQUETE.Req_pere*RAPPORT.Id_req
Il ne reste plus qu'à appeler le report REQUETE SANS RAPPORT pour visionner le résultat.

4.3 Editer un Rapport Général.

On commence par appeler la forme DEMANDE RAPPORT par le macro APPEL DEMANDE RAPPORT. Cette forme n'est reliée à aucune table et est juste là pour collecter des informations. Il faut ensuite définir un ensemble sur lequel on veut faire la recherche(il y en a un par défaut: toutes les requêtes). Il est possible de choisir: les requêtes rejetées, les requêtes non réalisées, etc. On indique ensuite les priorités recherchées (par défaut les 5). Il est possible d'entrer un nom de produit, une version, un environnement, un langage ou une fonctionnalité grâce à des combo box qui proposent la liste des différentes valeurs existantes mais qui ne permettent pas l'entrée de nouvelles valeurs. On peut d'ailleurs annuler la valeur d'une combo box en double cliquant, ce qui appelle un macro qui met la valeur de la combo box à nulle.

Il est possible de spécifier 2 bornes pour les requêtes recherchées. On peut demander toutes les requêtes après telle date+time et avant telle autre date+time. On dispose de la liste des identifiants de requêtes pour choisir ses valeurs. Dès que l'on sort d'une de ces 2 boites de dialogue, un macro: VERIFIDMINMAX est lancé. Il vérifie que les 2 dates ne sont pas les mêmes (si c'est le cas, il annule la date supérieure) et que la date inférieure n'est pas plus grande que la date supérieure (si c'est le cas, il copie la date supérieure dans le champs de la date inférieure et met la date supérieure à nul). A noter que l'on peut introduire une seule borne, l'autre étant nulle. On pourra ainsi demander les requêtes postérieures à telle date sans borne supérieure.

Une fois qu'on a entré des valeurs pour les champs, on peut lancer le macro SEARCH RAPPORT1 en cliquant sur Ok.
Le système commence par vérifier qu'il y a au moins une des priorités de sélectionnées sans quoi la recherche sera impossible, toutes les requêtes ayant une priorité par défaut.
Le système va mettre dans la table LISTE toutes les requêtes répondant aux critères choisis.

Tout d'abord, on sélectionne les requêtes faisant partie de l'ensemble choisi:
Voici les différents cas possibles:

- **Toutes les Requêtes**: le système lance le query ALL REQUETE OK qui copie toutes les requêtes de la table REQUETE dans la table LISTE puis lance la clause SQL suivante: insert into Liste select * from Requete_REJETEE, qui va ajouter les requêtes rejetées.

- **Toutes les Requêtes non Rejetées**: le système lance le query ALL REQUETE OK.

- **Les Requêtes à Analyser** : le système lance le query ALL REQUETE OK puis le delete query: REQUETE NON ANALYSEE1, qui efface de LISTE les requêtes qui ont un identifiant qui se trouve dans la table ANALYSE
  (joints: LISTE.Id_req*REQUETE.Id_req
  REQUETE.Id_req*ANALYSE.Id_req)
  On doit, comme d'habitude, effacer les requêtes fils qui sont analysées. Pour cela le delete query REQUETE NON ANALYSEE2 fait les joints: LISTE.Id_req*REQUETE.Id_req
  REQUETE.Id_req*GROUPE_REQUETE.Req_fils et
  GROUPE_REQUETE.Req_pere*ANALYSE.Id_req

- **Les Requêtes réalisées**: le système lance le append query REQUETE AVEC RAPPORT1 (joint: REQUETE.Id_req*RAPPORT.Id_req) qui ajoute dans LISTE les requêtes qui ont un rapport, sauf les requêtes fils. Le append query REQUETE AVEC RAPPORT2 s'en charge
  (joints: REQUETE.Id_req*GROUPE_REQUETE.Req_fils
  GROUPE_REQUETE.Req_pere*RAPPORT.Id_req)

- **Les Requêtes seulement Analysées**: le système lance 4 queries:
  REQUETE SEULEMENT ANALYSEES1 : le système ajoute dans LISTE les requêtes qui sont analysées, sauf les requêtes fils analysées
  (joint:REQUETE.Id_req*ANALYSE*Id_req)
  REQUETE SEULEMENT ANALYSEES2 : le système ajoute dans LISTE les requêtes fils analysées (joints; REQUETE.Id_req*GROUPE_REQUETE.Req_fils
  GROUPE_REQUETE.Req_pere*ANALYSE.Id_req)

  REQUETE SEULEMENT ANALYSEES3: le système efface de tout cet ensemble les requêtes qui ont un rapport (sauf les requêtes fils qui ont un rapport). Joint:
  LISTE.Id_req*RAPPORT*Id_req

  REQUETE SEULEMENT ANALYSEES4 :le système efface de LISTE les requêtes fils qui ont un rapport en faisant les joints: LISTE.Id_req*REQUETE.Id_req
  REQUETE.Id_req*GROUPE_REQUETE.Req_fils et
  GROUPE_REQUETE.Req_pere*RAPPORT.Id_req

- **Les Requêtes non Réalisées** : le système lance le query ALL REQUETE OK qui copie dans LISTE les records de REQUETES.
  Puis le système lance le delete query REQUETE SANS RAPPORT1, qui fait les joints : LISTE.Id_req*REQUETE.Id_req et REQUETE.Id_req*RAPPORT.Id_req. On efface ainsi toutes les requêtes qui ont un rapport, sauf celles qui sont Req_fils.
  C'est ce que fait REQUETE SANS RAPPORT2, en faisant les joints :
  LISTE.Id_req*REQUETE.Id_req, REQUETE.Id_req*GROUPE_REQUETE.Req_fils,
  GROUPE_REQUETE.Req_pere*RAPPORT.Id_req.

- **Les Requêtes rejetées**: le système copie dans LISTE tous les records de la table LISTE_REJETEE.

On a ainsi toutes les requêtes de l'ensemble choisi dans LISTE. Il se peut que cet ensemble soit vide, le système affiche alors un message et on arrête le macro là.

Si on n'a pas sélectionné toutes les priorités, le système doit effacer de LISTE les records qui n'ont pas les priorités choisies. Il teste alors les 5 boites à cocher et quand une à la valeur NO, on fait une clause SQL delete.
On vérifie ensuite que LISTE n'est pas devenue vide.

Pour chacun des champs suivants: Nom de Produit, Version, Langage, Environnement, Fonctionnalité, le système efface dans LISTE , si la valeur introduite n'est pas nulle, toutes les requêtes qui ont une valeur nulle ou différente de celle introduite.
Si on a demandé les requêtes postérieures à une date, on efface dans LISTE celles qui sont antérieures à cette date (même principe pour l'autre borne).

On a alors dans LISTE toutes les requêtes demandées. On peut alors ouvrir une autre forme: DEMANDE RAPPORT 2 qui va demander quels sont les critères suivant les mots-clé. le système vide la table MC_RECHERCHE qui va contenir les mots_clés recherchés.

Comme le montre la forme DEMANDE RAPPORT2, il est possible de faire 2 types de recherches. D'ailleurs, pour obtenir des informations à ce sujet, il suffit d'appeler le macro INFO RECHERCHE en cliquant sur une list box.

On a un groupe de bouton action qui permet de choisir le type de recherche:
Recherche relachée : le système sélectionne les requêtes qui ont au moins un des mots_clé choisi.
S'il n'y a pas de mots-clé, on sélectionne toutes les requêtes dans LISTE.
Recherche stricte : le système sélectionne les requêtes qui ont exactement les mots-clé choisis. Si on ne choisit pas de mots-clé, le système prend les requêtes sans mot-clé.
Dès qu'on change la valeur d'un bouton radio, le système appelle le macro TEST RECHERCHE.

En effet, le système doit tester si on viens de choisir une recherche stricte. Dans ce cas, le système copie la table LISTE dans LISTE TEMPORAIRE et vide la table MC_RECHERCHE qui contient les mots-clé sur lesquels se fait la requête (lors d'une recherche stricte, la table LISTE_TEMPORAIRE contient seulement les requêtes ayant les mots-clé déjà enregistrés.). Si par la suite on change d'avis et que l'on fait une recherche relachée, on pourra récupérer dans la table LISTE les requêtes sélectionnées avec les critères de la forme DEMANDE RAPPORT

Par la suite, quand on ajoutera un mot-clé (macro ADD MC RECHERCHE), après avoir testé que l'on n'a pas déjà enregistré le mot-clé, si on fait une recherche stricte, on appellera le append query RECHSTRICT PLUSIEURSMC1 qui sélectionnera dans LISTE_TEMPORAIRE les requêtes qui ont ce mot-clé et copiera les identifiants dans la table CHOIX_ID qui servait,

rappelons le, de source pour les combo box permettant de faire des groupes de requêtes. Le joint de ce query est LISTE_TEMPORAIRE.Id_req*MOT_CLE.Id_req.
Le système vide alors la table LISTE_TEMPORAIRE et lance le append query RECHSTRICT PLUSIEURSMC2 qui va copier dans LISTE_TEMPORAIRE les requêtes dont l'identifiant est dans CHOIX_ID (joint: LISTE.Id_req*CHOIX_ID.Id_req). Remarquons que l'on n'aurait pu effacer les requêtes qui n'avaient pas le mot-clé choisi directement de la table LISTE_TEMPORAIRE. En effet, une requête ayant le mot-clé choisi et un autre mot-clé aurait été effacée. On copie également le mot-clé désigné dans la table MC_RECHERCHE.

Si l'utilisateur fait une requête relachée, le système se contente de copier le mot-clé dans MC_RECHERCHEE.

Quand on efface un mot-clé (macro DEL MC RECHERCHE), si l'utilisateur fait une recherche relachée, le système se contente d'effacer le mot-clé de la table MC_RECHERCHE.
Si l'utilisateur fait une recherche stricte, le système doit effacer tous les mots-clé déjà sélectionnés et recommencer l'entrée de mots-clé. On vide alors la table MC_RECHERCHE (qui contient les mots-clé recherchés), et on recopie la table LISTE (qui contient toutes les requêtes ayant les critères de la forme DEMANDE RAPPORT) dans la table LISTE_TEMPORAIRE.

Quand on a entré tous les mots-clé, le système lance le macro SEARCH RAPPORT2.

Si l'utilisateur fait une recherche relachée et qu'on a enregistré des mots-clé, le système doit sélectionner dans LISTE les requêtes qui ont au moins un des mots-clé choisi.
Il lance alors l'append query RECH PLUSIEURS MC qui copie dans LISTE_TEMPORAIRE les records de REQUETE qui ont un des mots-clé de MC_RECHERCHE. (on fait les joints: LISTE.Id_req*MOT_CLE.Id_req et MOT_CLE.Mot_clé*MC_RECHERCHE.Mot_clé).

Si l'utilisateur fait une recherche stricte avec aucun mot-clé, le système doit sélectionner dans LISTE les requêtes qui n'ont aucun mot-clé. Le delete query RECH AUCUMMC s'en charge en faisant le joint LISTE.Id_req*MOT_CLE.Id_req.

Si l'utilisateur fait une recherche stricte avec mot-clé, quand on lance ce macro, on a dans la table LISTE toutes les requêtes sélectionnées avec les critères de la forme DEMANDE REPORT.
Dans la table LISTE_TEMPORAIRE, on a toutes les requêtes répondant aux critères de la forme DEMANDE RAPPORT et en plus qui ont au moins des mots-clé demandés. Mais le système doit effacer les requêtes qui ont d'autres mots-clé que ceux demandés.
Le système lance alors les queries suivants:

-MC NON RECH1: le système copie dans la table MC_NON_RECHERCHE tous les différents mots-clé existants.

-MC NON RECH2 : le système efface de la table MC_NON_RECHERCHE les mots-clé qui se trouvent dans la table MC_RECHERCHE.
(joint:MC_NON_RECHERCHE.Mot_CLÉ*MC_RECHERCHE.Mot_clé)
On a alors dans la table MC_NON_RECHERCHE tous les mots-clé non cherchés. Les requêtes sélectionnées ne doivent pas avoir un de ces mots-clé.

-RECHSTRICT PLUSIEURSMC3: le système efface alors de LISTE_TEMPORAIRE les requêtes qui ont un des mots-clé non cherché. Joints:
LISTE_TEMPORAIRE.Id_req*MOT_CLE.Id_req
MOT_CLE.Mot_clé*MC_NON_RECHERCHE.Mot_clé.
On a alors dans LISTE TEMPORAIRE toutes les requêtes qui ont exactement les mots-clé choisis, et seulement ceux-là. le système copie alors la table LISTE_TEMPORAIRE dans LISTE afin d'avoir toutes les requêtes sélectionnées dans la même table, pour tous les cas.

Le macro finit par tester si la table LISTE est vide. Dans ce cas, le système indique un message.
Sinon, la table LISTE est triée selon l'identifiant et on ouvre la forme CLASSEMENT.

En fait, la forme CLASSEMENT (read only) rappelle les critères choisit précédemment et comme elle est reliée à la table LISTE, elle permet de voir toutes les requêtes répondant aux critères choisis. On peut y choisir un ordre de classement pour le rapport.
Quand l'utilisateur clique OK, le système appelle le macro tri qui trie la table LISTE selon l'ordre choisi et vide les tables intermédiaires que l'on n'utilisent plus: LISTE_TEMPORAIRE, CHOIX_ID, MC_NON_RECHERCHE.

La table MC_RECHERCHE sert de source pour le rapport: REPORT GENERAL qui indique, à partir des formes DEMANDE RAPPORT et DEMANDE RAPPORT2 les critères choisis.
Le report header inclus un sous-rapport : LISTE MC RECHERCHE qui est associé à la table MC_RECHERCHE. l'utilisateur peut alors voir les mots-clé recherchés dans le report.